Cantor User Report
Version 2.0

W. C. Athas
C. L. Seitz

Computer Science Department
California Institute of Technology

5232:TR:86

Cantor User Report
Version 2.0

W.C. Athas
C.L. Seitz

Computer Science Department
California Institute of Technology
Pasadena, Calif. 91125

Technical Report 5232:TR:86
January, 1987

The research described in this report was sponsored in part by the Defense Advanced Research Projects Agency, ARPA order number 3771, and monitored by the Office of Naval Research under contract number N00014-79-C-0597, and in part by grants from Intel Scientific Computers and Ametek Computer Research Division.

Contents

List of Figures

Chapter 1

Introduction

Cantor is an object-oriented programming notation for describing concurrent computations in terms of small concurrent objects. The objects of Cantor are independent computing agents that interact only by message passing. Each object is comprised of a set of private variables, called the "persistent variables" or "acquaintances", a message list used to identify the contents of a message, and a sequence of commands that describes how the object will react when it receives a new message. All variables used in Cantor programs are dynamically typed and lexically scoped.

1.1 Cantor Objects

Normally a Cantor object is at rest and remains so until a message arrives for it. The response of an object to a message is determined by the contents of the message, the current contents of the persistent variables, and a sequence of procedural statements defining the fundamental "actions" that an object can perform. These actions include sending new messages and creating new objects. Other actions that an object can do include evaluating arithmetic expressions, assignment, and making decisions about what actions to perform next. An object cannot run indefinitely. After an object finishes processing a message, it must either ready itself to receive another message or self-destruct.

1.2 Object Interaction

Message passing in Cantor is based upon the sending object possessing a "reference" to the destination object. The message passing model that Cantor uses is based upon the model used by Cosmic Kernel [9], namely, messages exhibit arbitrary delay when travelling from sender to destination. This notion of reference is comparable to the notion of "pointer" found in programming languages such as C and Pascal, in that both pointer and reference are a *spatial* decoupling between the name of an object and the instance of the object. Cosmic Kernel and Cantor expand upon this interpretation by making reference also a *temporal* decoupling between the name of an object and its instance. Therefore, sending a message to an object is not an instantaneous application of the message to the destination object.

Cantor takes the temporal decoupling one step further by associating the decoupling not only with message passing, but also with object creation. Thus a reference to an object can exist and be computed on before the instance of the object exists. However, for an object to accept a message, the object must be instantiated.

The message passing model of Cantor deviates slightly from the model of Cosmic Kernel with respect to preservation of message order. In Cosmic Kernel, messages sent between two objects that are in direct communication will arrive in the same order as they were sent. Because Cantor objects can *become* other objects, an object is not necessarily always stationary in Cantor. The preservation of message order property is contingent upon not using the become statement.

1.3 What Cantor Does for the Programmer

The task of writing concurrent programs in Cantor focuses upon finding a good concurrent formulation for the intended application. The tasks of assigning objects to processing nodes and managing the delivery of messages to objects is handled jointly by the Cantor compiler and runtime system. Although the Cantor system manages these resources for the programmer, the choice of algorithms and programming style influences the performance of the computation. One of the objectives of this report is to suggest some programming paradigms that have been found to be useful and efficient. Hopefully many more such paradigms will be discovered as our experience with Cantor grows.

1.4 What Cantor Doesn't Do for the Programmer

Data and control structures common to many high-level programming notations are not fundamental to Cantor. For example, data structures such as arrays, records, queues, and stacks are not included. These data structures must be built explicitly out of Cantor objects. With respect to control structures, iteration is not included as one of the basic actions. Instead, structured programming constructs such as `for` and `while` must be programmed by message passing operations.

Programmers accustomed to the well-known imperative programming languages used on sequential processors may find programming in Cantor challenging, i.e. difficult, at first. Data structures are described as compositions of objects and control structures are described as sequences of message passing actions. These data and control structures are not prepackaged as part of Cantor. However, the programmer can accumulate libraries of object definitions for particular problems.

Cantor bears some kinship to LISP, with the Cantor object replacing the "cons" cell as the atomic constructor.

Chapter 2

The Structure of Cantor Programs

The purpose of this chapter is to explain the syntax and semantics of Cantor programs. Our strategy is to introduce the various syntactic constructs both by their BNF descriptions and with examples of their use. A complete BNF definition for Cantor is given in Figure 2.1. The examples were chosen for their ability to illustrate in simple contexts the various constructs used in Cantor programs. They were not chosen for their usefulness nor efficiency. The *de luxe* programming examples in Chapter 3 are more nearly representative of practical computations.

2.1 Introductory Examples

For these introductory examples, we shall emphasize the overall structure of a Cantor program, the way in which Cantor objects are defined and instantiated, and some of the ways in which iteration and recursion can be expressed.

The structure of a Cantor program is reflected in the first three rules of the BNF specification:

⟨*program*⟩ ⟹ ⟨*object definition*⟩* ⟨*description*⟩
⟨*object definition*⟩ ⟹ ⟨*object name*⟩ ⟨*persistent list*⟩ : : ⟨*description*⟩
⟨*description*⟩ ⟹ (*[| [ ) ⟨*body*⟩]

Characters that are in boldface (darker) or words that are underlined denote tokens that are part of Cantor's syntax. For example, the asterisk following ⟨*object definition*⟩ in the rule for ⟨*program*⟩ denotes the "Kleene star" as it is used in BNF. The boldface asterisk in the ⟨*description*⟩ rule denotes the Kleene star as it is used in Cantor.

To apply these BNF rules, consider this simple but complete Cantor program:

```
[ (console) send ("Hello World") to console ]
```

In time-honored tradition, this first example program prints "Hello World" on the programmer's output device. As shown in the BNF rule for ⟨*program*⟩, a Cantor program consists of repeated ⟨*object definition*⟩s, including none as in this case, followed by a ⟨*description*⟩.

The ⟨*description*⟩ here is a special object definition called the **main** object. When a Cantor program starts executing, the **main** object is the only object present, and this object starts to execute only after receiving a message from the runtime system hosting the computation. To understand the meaning of this single line, the BNF rule for ⟨*body*⟩ needs to be expanded:

⟨*program*⟩ ⟹ ⟨*object definition*⟩* ⟨*description*⟩

⟨*object definition*⟩ ⟹ ⟨*object name*⟩ ⟨*persistent list*⟩ : : ⟨*description*⟩

⟨*description*⟩ ⟹ (*[| [ ) ⟨*body*⟩]

⟨*body*⟩ ⟹ ⟨*sequence*⟩ | ⟨*case*⟩ | ⟨*description*⟩+

⟨*sequence*⟩ ⟹ ⟨*message list*⟩ ⟨*statement*⟩*

⟨*case*⟩ ⟹ **case** (⟨*variable name*⟩) **of** ⟨*case entry*⟩+

⟨*case entry*⟩ ⟹ ⟨*selector*⟩ : ⟨*sequence*⟩

⟨*statement*⟩ ⟹ ⟨*if*⟩ | ⟨*let*⟩ | ⟨*send*⟩ | ⟨*assign*⟩ | ⟨*control*⟩ | ⟨*description*⟩

⟨*if*⟩ ⟹ **if** ⟨*expression*⟩ **then** ⟨*statement*⟩+ { **else** ⟨*statement*⟩+ } **fi**

⟨*let*⟩ ⟹ **let** ⟨*variable name*⟩ = ⟨*expression*⟩

⟨*send*⟩ ⟹ **send** ⟨*list*⟩ **to** ⟨*expression*⟩

⟨*assign*⟩ ⟹ ⟨*name*⟩ :- ⟨*expression*⟩

⟨*control*⟩ ⟹ **exit** | **repeat** | **become** ⟨*expression*⟩

⟨*persistent list*⟩ ⟹ ⟨*name list*⟩

⟨*message list*⟩ ⟹ ⟨*name list*⟩

⟨*name list*⟩ ⟹ () | (⟨*name*⟩ { , ⟨*name*⟩ }*)

⟨*expression*⟩ ⟹ ⟨*expr1*⟩ { (**or** | **xor**) ⟨*expr1*⟩ }*

⟨*expr1*⟩ ⟹ ⟨*expr2*⟩ { **and** ⟨*expr2*⟩ }*

⟨*expr2*⟩ ⟹ ⟨*expr3*⟩ { (= | <> | < | > | <= | >=) ⟨*expr3*⟩ }*

⟨*expr3*⟩ ⟹ ⟨*expr4*⟩ { (+ | -) ⟨*expr4*⟩ }*

⟨*expr4*⟩ ⟹ ⟨*primitive*⟩ { (* | / | **mod**) ⟨*primitive*⟩ }*

⟨*primitive*⟩ ⟹ ⟨*variable name*⟩ | ⟨*selector*⟩ | ⟨*reference*⟩ | ⟨*real*⟩ |
abs ⟨*primitive*⟩ | **not** ⟨*primitive*⟩ | (⟨*expression*⟩)

⟨*selector*⟩ ⟹ ⟨*integer*⟩ | ⟨*logical*⟩ | ⟨*symbol*⟩

⟨*reference*⟩ ⟹ **self** | ⟨*object name*⟩ ⟨*list*⟩

⟨*list*⟩ ⟹ () | (⟨*expression*⟩ { , ⟨*expression*⟩ }*)

⟨*symbol*⟩ ⟹ " ⟨*char*⟩* "

⟨*logical*⟩ ⟹ **true** | **false**

⟨*name*⟩ ⟹ ⟨*ident*⟩

Figure 2.1: BNF Description for Cantor

⟨*body*⟩ ⟹ ⟨*sequence*⟩ | ⟨*case*⟩ | ⟨*description*⟩$^+$
⟨*sequence*⟩ ⟹ ⟨*message list*⟩ ⟨*statement*⟩*
⟨*case*⟩ ⟹ case (⟨*name*⟩) of ⟨*casebody*⟩$^+$

The body of an object definition is either a message list followed by a sequence of zero or more statements, a case construct, or a series of nested descriptions. The case construct will not be used in these introductory examples. For the main object of the "Hello World" program, the body is an instance of the ⟨*sequence*⟩ rule, comprised of a message list, containing a single component named console, followed by a send statement.

The general interpretation of a ⟨*description*⟩ is this: Whenever an executing object encounters an open square bracket, i.e. a new description, execution of the object stops until a new message arrives for it. Initially an object starts executing at the outermost left bracket, and is therefore initially waiting for a message. When a message arrives it is copied into the message list of the body of the description. The object then executes statements until the outermost close square bracket is encountered, or until another description is reached.

For the main object, the message list contains references to external objects that are passed into the Cantor program by the runtime system hosting the program. For the "Hello World" program, the message from the outside contains a reference to an object internally named console. A message sent to the reference contained in console will cause the message contents to be displayed on some output device.

A statement can change the content of an internal variable, alter the control flow within the object, or cause an external effect by sending a message or creating a new object. The entire set of possible statements is given by the BNF for a ⟨*statement*⟩:

⟨*statement*⟩ ⟹ ⟨*if*⟩ | ⟨*let*⟩ | ⟨*send*⟩ | ⟨*assign*⟩ | ⟨*control*⟩ | ⟨*description*⟩

All of these types of statements will appear in the examples in this section.

The main object in our example contains only a single statement, a send statement. The syntax for the send statement is the following:

⟨*send*⟩ ⟹ send ⟨*list*⟩ to ⟨*expression*⟩
⟨*expression*⟩ ⟹ ... ⟨*primitive*⟩ ...
⟨*primitive*⟩ ⟹ ⟨*variable name*⟩ | ⟨*reference*⟩ ...
⟨*reference*⟩ ⟹ self | ⟨*object name*⟩ ⟨*list*⟩

The meaning of the send statement is to copy the contents of ⟨*list*⟩ into a message and then send the message to the object whose reference is the value of ⟨*expression*⟩.

Lists have the following syntax:

⟨*list*⟩ ⟹ () | (⟨*expression*⟩ { , ⟨*expression*⟩ }*)

Expressions in Cantor can take on a variety of forms as directed by the precedence structure of the operators built into Cantor. However, all expressions must evaluate to one of the five primitive data types that are built into Cantor. The five types of primitive values are:

- reference, the address of another object.
- integer, e.g. 42.
- real number, e.g. 3.14159265.
- symbol, e.g. "Hello World".
- logical (Boolean), viz. true or false

All Cantor values are tagged with type information. When the contents of a list are copied into a message, each expression is evaluated to yield a value, and these values, each tagged with its type, make up the contents of the message.

For the ⟨*send*⟩ rule, the destination of the message is denoted by an ⟨*expression*⟩ that must evaluate to a reference value. Any other value type is a programming error. For our first example, here repeated for convenience in reading:

[(console) send ("Hello World") to console]

console must evidently be a reference, "Hello World" is a literal symbol, and the effect of the single send statement in the body of the description is to dispatch this message to console. At the object referred to as console, the message is identified as a list containing one symbol, and this symbol is printed.

Referring again to the BNF, a reference value can in general be the value associated with a variable name, e.g. a component of the persistent list for the object or a component of the message list for the current description, or can be introduced by the rule for ⟨*reference*⟩. This rule states that a reference can be produced by the keyword self, which produces a reference value for the object where the keyword self occurs, or can be manufactured by composing the name of an object definition with a list. The effect of this composition is to create a new instance of the object and to "install" the list into the persistent list of the new object.

In order to illustrate how objects create other objects, let us send the "Hello World" message followed by a "Goodbye Cruel World" message via a relay object:

```
relay(destination) :: [(message) send (message) to destination]

[ (console)
  send ("Hello World") to console
  send ("Goodbye Cruel World") to relay(console)
]
```

The definition of the relay object includes the single persistent variable named destination, which by its use in the send statement must be a reference. When the main object refers to relay, it creates an instance of the relay object, initializes its persistent variable with the value of console, and obtains a reference to the new object. When the message arrives for the relay object, it is copied into the list (message), and the send statement causes this same list to be dispatched to the destination that had been "installed" in the object when it was created.

Incidentally, because messages traveling on different paths may encounter different delays, it is quite possible that the two messages will arrive at and be printed by the console object in the reverse order from what you might expect.

In this example we happen to have passed the intended destination to each new relay object by "installing" it initially as an persistent variable, and passed the message we wanted the object to relay in the message to it. One could equally have written the program to install the message and to send the destination. One could also avoid using any persistent variables at all in the relay, as in:

```
relay() :: [ (where,what) send (what) to where]

[ (console) send(console,"Hello World") to relay()]
```

or one could include both the destination and what to print as persistent variables:

```
relay(where,what) :: [() send (what) to where]

[ (console) send () to relay(console,"Hello World")]
```

Persistent variables hold state information that persists from one message and object execution to the next. In all of the examples above the object "self-destructs" when the execution reaches the close square bracket, so the objects do not require persistent variables.

One way to define an object so that it is persistent is to start its description with *[instead of [. This form specifies that the description replaces itself with itself when the corresponding right square bracket is encountered. If one wanted to be stingy in the number of relay objects created , one could define a persistent relay object, create only one instance, and use it over and over:

```
relay(where) :: *[(message) send (message) to where]

[ (console)
  let  r = relay(console)
  send ("Hello World") to r
  send ("Goodbye Cruel World") to r
]
```

This example also introduces the let statement, which has the syntax:

⟨*let*⟩ ⟹ let ⟨*variable name*⟩ = ⟨*expression*⟩

The let statement is useful for establishing a placeholder for the value of an expression and for introducing local, persistent variables into a description. Cantor variable names can originate either from the persistent list, in which case the variables are global to the object, or from the message list or a let statement inside either the current description or one of

the lexically enclosing descriptions. The value of the expression part of the let statement of this example is the reference to the single relay object created. This reference is bound to the variable r, which is used to specify the destination for both of the send statements within the main object.

It happens that when this computation is completed, as indicated by there being no object processing messages and no message left in the system to process, the instance of the relay object still persists. Not to worry. The completion of the computation is detected by the absence of messages, and more generally, such an object is automatically determined to be "irrelevant" and is collected as "garbage" when no other object is able to send it a message.

Another form for specifying a replacement description is expressed by nested descriptions. For example, in this variation on the previous example:

```
noisy_relay(where) ::
*[ (message) send (message, "odd") to where
   [(message) send (message, "even") to where ]
 ]

 [ (console)
   let  r = noisy_relay(console)
   send (1) to r
   send (2) to r
   send (3) to r
 ]
```

the noisy_relay object appends a second element to the message list as it is relayed, indicating whether it is an odd- or even-numbered message.

If you were now to start writing Cantor programs, you might well complain that Cantor lacks iteration and procedure constructs. Iteration internal to an object would allow the object to execute for an indefinite period, which is contrary to Cantor's computational model. Program behaviors corresponding to what appears in other programming notations as iteration and procedures are in any case easily expressed in terms of Cantor's message sending and object creation mechanisms.

Let us start with idioms for iteration in Cantor. The following example program:

```
repeat_relay(where,what)::
[(i) if i>0
        then send (what) to where
             send (i-1)  to repeat_relay(where,what)
     fi
]

[(console) send (5) to repeat_relay(console,"Hello World")]
```

prints "Hello World" a number of times determined by the message sent by the main object, five times in this case. The iteration is accomplished by repeat_relay objects that create the necessary number of instances of repeat_relay objects. This program also introduces the if statement, which has the syntax:

$$\langle if \rangle \implies \underline{\text{if}}\ \langle expression \rangle\ \underline{\text{then}}\ \langle statement \rangle^{+}\ \{\ \underline{\text{else}}\ \langle statement \rangle^{+}\ \}\ \underline{\text{fi}}$$

and has the conventional meaning.

It is more natural and efficient to make repeat_relay persistent, and have it send a message to itself, rather than to create a chain of instances:

```
repeat_relay(where,what) ::
*[(i) if i>0 then send (what) to where
                  send (i-1)  to self
      fi
 ]

 [(console) send (5) to repeat_relay(console,"Hello World")]
```

This basic idiom for iteration leaves an instance of the repeat_relay object after the iteration is complete. However, one can explicitly override the Kleene star by using the ⟨*control*⟩ construct exit, as in:

```
repeat_relay(where,what)::
*[ (i) if i>0
          then send (what) to where
               send (i-1)  to self
          else exit
      fi

 ]

 [(console) send (5) to repeat_relay(console,"hello")]
```

Alternatively, one can define the repeat_relay object with no replacement and use the repeat construct:

```
repeat_relay(where,what) ::
[(i) if  i>0
        then send (what) to where
             send (i-1)  to self
             repeat
      fi
]

[(console) send (5) to repeat_relay(console,"hello")]
```

These two programs have exactly the same behavior. The exit statement can be thought of as overriding the *[...] of the description in which it is contained, and the repeat statement the [...]. Using exit inside of a [...] is harmless, though aesthetically displeasing, likewise for using repeat inside of a *[...].

The final group of examples of this section is meant to illustrate that in Cantor, just as in other programming notations, one can compute the overused example of the factorial function in numerous ways, either by a linear iterative process or by a linear recursive process. Also, we will define the **factorial** object in each example such that it is invoked by a consistent convention that is similar to the way in which functions are called in conventional programming notations.

First, here is a recursive definition of a **factorial** object and a **main** object to "call" it:

```
factorial(reply, n)::
[ ()
  if n<2
     then send (1) to reply
     else send () to factorial(self, n-1)
          [(m) send (n*m) to reply ]
  fi
]

[(console) send () to factorial(console, 6)]
```

The occurrences of the ⟨*object name*⟩ **factorial** are concatenated with a ⟨*list*⟩ whose first element is the object to which to send the answer, and whose remaining elements are similar to the argument list of a function. Since all the information necessary to the computation of the factorial is conveyed in this list, the message list is empty.

The definition of the **factorial** object follows almost directly from the recursive definition. The use of a replacement description in the else clause serves to receive the message result of the computation of $(n-1)!$. Factorial is a linear recursive function. It necessarily consumes resources linear in its argument in building a stack of decreasing values of n. The stack in this example is built from instances of the **factorial** object itself. If it seems that one is carrying too much baggage with a stack of **factorial** objects, one can create a stack of more elementary objects by factoring out the nested description of:

[(m) send (n*m) to reply]

into a separate object definition called **mult**. The instances of the object **mult** will keep the stack values linked together, freeing the single instance of the **factorial** object on each recursion, as in:

```
mult(reply, n)::[ (m) send (n*m) to reply]

factorial(reply, n)::
[ ()
  if n<2
     then send (1) to reply
     else reply := mult(reply,n)
          n := n-1
          send () to self
          repeat
  fi
]

[(console) send () to factorial(console, 6)]
```

This example is the first to use assignment, which in Cantor has the conventional interpretation. The iterative behavior of the factorial object builds a chain of mult objects with decreasing values of x, the first with a reference to the object expecting the answer, and the rest with reference to its immediate predecessor. When the iteration terminates the mult objects generate the product. In order to generate the product as the decreasing values are produced, one would need another variable, and would have converted this program to the iterative version of computing the factorial function:

```
factorial(reply, n)::
[ let m = 1
  send () to self
  [ () if n<2
          then send (m) to reply
          else m := m*n
               n := n-1
               send () to self
               repeat
       fi
  ]
]

[(console) send () to factorial(console, 6)]
```

The additional variable m is introduced and initialized by the let statement. The nested description does the actual work.

2.2 Expressing Recursive Functions in Cantor

The Cantor examples for the factorial function have demonstrated some of the ways linear recursive functions can be written in Cantor. For an example of another type of recursion, called tree recursion, consider the task of computing the n^{th} Fibonacci number by the well-known recursive formula:

$$\begin{aligned} fib_0 &= 0 \\ fib_1 &= 1 \\ fib_n &= fib_{n-1} + fib_{n-2} \end{aligned}$$

A Cantor program to compute this recursive function is shown in Figure 2.2. The program begins by creating a single instance of the fib object and then sends the new object the message (console, 15). The Fibonacci function is a side-effect free ("pure") function. In keeping with this property, the Cantor version uses no persistent variables.

The fib object expects messages consisting of two components, a variable named caller, which is expected to be a reference value, and n, the argument of the function. Once the Fibonacci number for n has been computed, the answer is replied to caller.

If the value of n sent to fib is 0 or 1, then the value of n is replied to caller, otherwise the recursive step must be taken. Initiating the recursive invocation is straightforward. Two new fib objects are created and sent messages of n-1 and n-2. The question then arises where the two reply values should be sent. Clearly sending the two reply values directly to caller is incorrect because caller is expecting a single message containing the sum from the two recursions.

The return from the recursion therefore consists of receiving the two reply values, summing them together, and sending the sum to caller. To accomplish this set of actions, a third object, called adder, is introduced. The adder object is similar to the mult object used for the factorial program except that it waits for two messages instead of one. The adder object waits for the two reply values from the two fib objects, sums the two reply values, and then sends the sum to caller. The reference value of the caller variable is passed from the fib object to the adder object when fib creates the adder object.

The tree recursion formulation of the Fibonacci function generates two fib objects for each message received unless the integer value contained in the message is 0 or 1. Thus the program of Figure 2.2 generates a ridiculous number of objects. Due to the tremendous concurrency generated, the program is arguably useful since it provides a simple way of testing whether the runtime environment contains a concurrent machine or not.

A first improvement to the Fibonacci program would be to reduce the number of new objects created by the fib object. One possible reduction would be to modify the fib object shown in Figure 2.3. The description is made persistent and one of the two recursive message is sent to self while the other is still sent to a new fib object.

By sending messages to self, the number of new objects created is appreciably reduced. The use of the keyword exit to complement the asterisk in front of the description ensures that all of the objects are eventually disposed of. This second version does not, however, reduce the number of messages generated.

```
adder (caller) :: [(x) [ (y) send (x + y) to caller ] ]

fib () :: [ (caller, n)
            if n = 0 or n = 1
               then send (n) to caller
               else let  a = adder(caller)
                    send (a, n-2) to fib()
                    send (a, n-1) to fib()
            fi ]

[ (console) send (console, 15) to fib() ]
```

Figure 2.2: Fibonacci Program

```
adder(caller) :: [(x) [ (y) send (x + y) to caller ] ]

fib () :: *[ (caller, n)
              if n = 0 or n = 1
                 then send (n) to caller
                      exit
                 else let  a = adder(caller)
                      send (a, n-2) to self
                      send (a, n-1) to fib()
              fi ]

[ (console) send (console, 15) to fib() ]
```

Figure 2.3: Fibonacci Program Using Less Objects

The Fibonacci programs of Figures 2.2 and 2.3 are poor concurrent formulations, because work is often duplicated needlessly. To see this duplication, consider the first three recursion levels of fib for $n = 15$:

$$\begin{aligned} fib_{15} &= fib_{13} + fib_{14} \\ fib_{14} &= fib_{12} + fib_{13} \\ fib_{13} &= fib_{11} + fib_{12} \end{aligned}$$

Thus fib_{13} is computed twice as is fib_{12}. This duplication of work repeats recursively, terminating with fib_0 and fib_1 being computed 987 times.

There are several solutions to this problem including a simple linear time, constant space, iterative solution for Fibonacci numbers. A solution that focuses not on the mathematical properties of Fibonacci numbers but rather on the behavior of the recursion is to introduce a "memo" function [1]. The idea is simply to maintain a table that maps values of n into the corresponding n^{th} Fibonacci number. Instead of having the fib object always send messages to two new instances of fib objects, the requests for the two new values of n are sent to a look-up table. If the values of n already exist in the table, the answers are replied to the adder objects as before. If the request for n is stored in the table but not the answer, the requests for the same value of n are chained together. Otherwise, the request is made into a new table entry, a new fib object is created and the value of n is sent to the new object.

A program to build the table in conjunction with the Fibonacci function is shown in Figure 2.4. The memo object is structured around the case construct, which is the alternate form for the body of a description. Each memo object initially contains a value for n called val in the persistent list of memo and a single requester, which is the caller from either a fib object or the main object. Each memo object can accept two "types" of messages, either a reply from a fib or adder object containing the value of the Fibonacci number for val, or a request to compute the Fibonacci number of val. To distinguish the different types of messages, the messages are "tagged" by the symbols "request" and "reply" respectively. Symbols are one of the primitive values in Cantor, and are distinguished by a string of characters enclosed in double quotes. The symbols of Cantor should not be confused with the notion of strings as found in other languages. Symbols in Cantor are only programmer defined constants that can be tested for equality.

The BNF description for the case construct is:

$$\begin{aligned} &\langle case \rangle && \Longrightarrow && \underline{\text{case}}\ (\ \langle variable\ name \rangle\ )\ \underline{\text{of}}\ \langle case\ entry \rangle^{+} \\ &\langle case\ entry \rangle && \Longrightarrow && \langle selector \rangle\ :\ \langle sequence \rangle \\ &\langle selector \rangle && \Longrightarrow && \langle integer \rangle\ |\ \langle logical \rangle\ |\ \langle symbol \rangle \end{aligned}$$

The case construct is a syntactic convenience for programming objects based upon message types. The chief reason for the inclusion of the case construct is to facilitate the tagging of messages, as for example in the memo object. The use of the tag provides a mechanism to vary the interpretation of the contents of a messages based upon the value of the first component of the message. The tag also serves the role of the "method" or "attribute" selector used in object-oriented programming. The tag of the message is used to invoke an attribute of an object by dispatching on the tag to one of the case entries. Each case entry corresponds to a method of the object. The first variable of the message is the method selector. The ⟨*selector*⟩ is enclosed in parentheses and follows the case keyword. The value of ⟨*selector*⟩ must be either a symbol, integer, or logical. This value is used to select which

```
adder (caller) :: [(x) [ (y) send ("reply", x + y) to caller ] ]

chain (item, next) :: [ (answer) send (answer) to item
                        send (answer) to next ]

memo (val, next, requesters) ::
*[ case (cmd) of
   "request" : (n, caller)
               if n = val
                  then requesters := chain(caller, requesters)
                  else if next = "nil"
                          then next := memo(n,"nil",caller)
                               send (n, next) to fib()
                          else send ("request", n,caller) to next
                       fi
               fi
    "reply" : (result)
               send (result) to requesters
               *[(discard, n, caller)
                 if n = val
                    then send (result) to caller
                    else if next = "nil"
                            then next := memo(n,"nil",caller)
                                 send (n, next) to fib()
                            else send ("request", n,caller) to next
                         fi
                 fi ]
]


fib () ::
[ (n, table)
  if (n = 0) or (n = 1)
     then send ("reply", n) to table
     else let  a = adder(table)
          send ("request",n-1,a) to table
          send ("request",n-2,a) to table
  fi ]

[ (console)  send (15, memo(15,"nil",console)) to fib() ]
```

Figure 2.4: Fibonacci Function with Look-Up Table

of the case entries that will be executed in response to the message. If none of the selectors match the value of the first message variable, then the message is discarded. If a selector matches the value of the first message variable, then the message list and sequence of statements following the selector is executed. Thus the message list for the case construct is separated into two parts. The first message variable is enclosed in parentheses following the case keyword while the remainder of the message list follows the selector and has the same syntax as the ⟨*sequence*⟩.

The case construct is analogous to the variant record as defined in the programming language Pascal [8].

For the memo object, the case construct is used to distinguish the two types of commands that can be sent to a memo object, either a request to compute a Fibonacci number or the reply from an adder or fib object with the answer for the Fibonacci number of val. The first component of a message accepted by a memo object is cmd. If cmd is equal to the symbol "request", then the remainder of the message is interpreted as a value of n to be computed on, and a reference value identifying the object that should receive the result. For the case of "request", the value of n is compared to that of val. If they are equal then the reference value of the message variable caller is enqueued by storing it as an acquaintance of a chain object. If val is not equal to n, then the "request" message is forwarded to the next memo object. If no next memo object exists, a new memo object is created along with a new fib object to compute the value of the Fibonacci number for n.

If the value of cmd is "reply", then the second message variable is the result of a previous request. The value is kept in result and replied to all of the objects that have requested an answer. Since the memo object expects a single "reply" message, all messages accepted after the "reply" message must be "request" messages. The replacement description, once a "reply" message has been accepted, is to check whether the request is for val. If yes then the result is sent to caller; otherwise the request is forwarded to the next memo object as before.

2.3 An Example Using a Vector

Cantor has no notion of aggregate data except of course for the Cantor object. Thus the look-up table for the Fibonacci function was implemented as a linked list of memo objects. As an example of using linked lists of objects for problems involving array indexing, consider the task of calculating the inner product of two vectors A and B, written $C = A \cdot B$. A program to calculate the inner product of the two vectors $A = [1,2,3]$ and $B = [4,5,6]$ is shown in in Figure 2.5.

The two vectors A and B are represented as linked lists of element objects. The element objects are not created "dynamically" but rather by a series of let statements in the main object. The first element object of each list has reference to the second element object, and the second element has reference to the third. The semantics of Cantor permit this leniency in evaluation only in a contiguous sequence of let statements and only when the values of the expressions are references. A program fragment of the form:

```
let x = y + 2
let y = x + 3
```

will yield unpredictable results because x and y cannot be reference values. Cantor's lenient interpretation of the evaluation order for contiguous let statements permits objects to possess mutual reference such as:

```
let x = f(y)
let y = f(x)
```

and also "self-loops" such as:

```
let x = f(x)
```

Aside from this new usage of the let statement, the interpretation of the program of Figure 2.5 is straightforward. To summarize, the cnode object is used to gather both the value and next link of A_i and B_i. Vectors A and B are expected to be of equal length. If one of the two next links is "nil", then the product of the two values of the two elements is sent to caller. The reader can check this sequence of events by setting the next link of A1 and B1 both to "nil". If the next link is not "nil", then a new cnode and adder object are created. The adder object is sent the product of the two values received from A_i and B_i, and the new cnode object is programmed to reply to the adder object by setting its persistent variable caller to the reference value of the adder object. The adder object is identical to the one used in the Fibonacci examples.

A possible refinement to this program is shown in Figure 2.6. The product step of cnode is factored out into a new object called mpyer. This new object is the same as adder except that it computes a product instead of a sum. The definition of element is changed so that the values are sent to the mpyer object and the next links are sent to the next cnode object. The caller acquaintance of the mpyer object is set to the adder object, thereby performing the multiply and add step of the inner product function outside of the cnode object.

The sending of two messages instead of one from the element object introduces concurrency that was not exploited in the program of Figure 2.5. The product step for the current index and the set-up for the next index are performed concurrently instead of being performed sequentially by the cnode object.

```
element (val, next) :: [(c_next) send (val.next) to c_next ]

adder (caller) :: [(x) [ (y) send (x + y) to caller ] ]

cnode (caller) ::
[ (v1, n1)
  [ (v2, n2)
    if n1 = "nil" then send (v1*v2) to caller
                  else let a = adder(caller)
                       let c = cnode(a)
                       send (v1*v2) to a
                       send (c) to  n1
                       send (c) to  n2
    fi
  ]
]

[ (console)
  let A1 = element(1,A2)  let A2 = element(2,A3)  let A3 = element(3,"nil")
  let B1 = element(4,B2)  let B2 = element(5,B3)  let B3 = element(6,"nil")

  let c = cnode(console)
  send (c) to A1
  send (c) to B1
]
```

Figure 2.5: Inner Product Program

```
element (val, next) :: [ (c_next,mpy)
                         send (next) to c_next
                         send (val)  to mpy
                       ]

adder (caller) :: [(x) [ (y) send (x + y) to caller ] ]

mpyer (caller) :: [(x) [ (y) send (x * y) to caller ] ]

cnode (caller) ::
[ (n1)
  [ (n2)
    if n1 = "nil"
       then send (0) to caller
       else let a = adder(caller)
            let m = mpyer(a)
            let c = cnode(a)
            send (c,m) to  n1
            send (c,m) to  n2
    fi
  ]
]

[ (console)
  let A1 = element(1,A2)  let A2 = element(2,A3)  let A3 = element(3,"nil")
  let B1 = element(4,B2)  let B2 = element(5,B3)  let B3 = element(6,"nil")

  let c = cnode(console)
  send (A1) to  c
  send (B1) to  c
]
```

Figure 2.6: Inner Product with Multiply Object Program

2.4 Building Data Structures in Cantor

The preceding programming examples have only touched upon some of the approaches for incorporating data structures into Cantor programs. Programs are often based upon organizing objects in terms of stacks, tables, arrays, etc., and then customizing the objects for the particular application, thereby blurring the distinction between the pure data structure and its realization within a program. Therefore, to provide a starting point for understanding the use of data structures in Cantor, four common data structures, stacks, queues, vectors, and two-dimensional arrays, are examined in their purest form.

2.4.1 Stacks

The object definition shown in Figure 2.7 behaves as an element for a stack. Two assumptions about the stack object are that each object is created containing a value and secondly, that there will never be more pops than pushes. The stack object has two "modes" of operation. The "full" mode occurs when the stack object contains a value for the stack. For each "push" message received, the object checks the value of the next variable. If the value of next is "nil", then a new stack object is created containing the current stack value, and the value contained in the "push" message is assigned as the content of the current stack element. If the value of next is a reference, then a "push" message is sent to next with an argument of content. The value contained in the received "push" message is then assigned to content.

When the first "pop" message is received, the stack object sends the value of content to caller and then enters the "empty" mode of operation. All "pop" messages received by a stack object that is in the empty mode are forwarded to next. If next contains the value "nil", then the symbol "empty" is sent to caller. The symbol "empty" will only be sent if more "pop" messages are sent to the stack than "push" messages.

When a "push" message is received by a stack object that is in the empty mode, the content of the message is assigned to content and the stack object returns to the full mode.

This formulation for a stack has two attractive properties. Multiple messages can be sent to the stack without having to synchronize with reply messages reply. For example, a series of "push" messages followed by a series of "pop" messages will behave correctly, independent of message delays.

A second property is that an interleaved series of "pop" and "push" messages will only involve the "top of stack" object. The first "pop" message will put the top of stack object into the empty mode. The following "push" message will place the top of stack object back into the full mode. For this formulation, the number of objects involved in a push or pop operation is limited to the difference between the number of "push" and "pop" messages processed by the stack.

One possible refinement to the stack object is to have the objects self-destruct if their value is popped and the value of next is "nil". The difficulty with this refinement is that the top of stack object has to be treated differently; otherwise the entire stack will self-destruct.

```
stack (content, next) ::
*[ case (cmd) of
   "push" : (val)
            if next = "nil"
               then next := stack(content, "nil")
               else send ("push",content) to next
            fi
            content := val
   "pop" : (caller)
           send (content) to caller
           *[ case (cmd) of
             "push" : (val)
                      content := val
                      exit
             "pop"  : (caller)
                      if next = "nil"
                         then send ("empty") to caller
                         else send ("pop", caller) to next
                      fi
            ]
 ]
```

Figure 2.7: Stack Object

2.4.2 Queues

Figure 2.8 depicts one formulation for expressing a queue in Cantor. The two operations performed on the queue object are advance queue front, returning the value at the head of the queue, and insert a new value at the tail of the queue. Although the stack example required only a single "top of stack" reference to be maintained by another object, queues require two references, one for the head of the queue and the other for the tail.

The queue is partitioned into two object definitions: queue_cell and queue_master. The queue_cell objects are used to form a linked list of the values stored in the queue. The queue_master object is used to control access to the linked list of queue_cell objects. The queue permits concurrent writes into the queue by sending the queue master object multiple "insert" messages. However, the queue should have only a single reader, i.e. the environment guarantees that only one object at a time will send "advance" messages to the queue_master object, and that the reply to the "advance" message is processed before another "advance" can be sent.

The outer case construct of the queue_master object waits for either type of message. If an "advance" message arrives first, then the next message received must be an "insert" message. For this case, no queue_cell objects are needed. The queue_master object waits for the "insert" message and then replies the value of the "insert" message to the object that sent the "advance" message.

Once the number of "insert" messages received exceeds the number of "advance" messages received, a queue_cell object is allocated and assigned to the local variables hd and tl. The inner case construct is then used to process messages. The receipt of an "insert" message results in a new queue_cell object. This new object is to be added at the tail of the current queue, a "link" message containing the value of the new tail object is sent to the old tail object and the variable tl is assigned the reference value of the new queue_cell object.

When a "advance" message is received, the value at the front of the queue must be retrieved and the front of the queue advanced. Fetching the value at the front of the queue and the value for the new queue front is accomplished by sending a "get" message to hd. From this point, two types of messages can be received: additional "insert" messages and the reply from hd. A third case construct is used to handle the two message types. The case for "insert" performs as before. The case for "hd reply" sends the value at the queue front to the object that sent the "advance" message, and assigns the new queue front value to hd. The receipt of the "hd reply" messages returns control to the second case construct. If the value of the head of the queue is "nil", then the queue is empty and control returns to the outer case construct.

```
queue_cell(value, next) ::
*[ case (cmd) of
   "link" : (new_next) next := new_next
   "get"  : (caller) send ("hd reply",next,value) to caller
                     exit
 ]


queue_master() ::
*[ case (cmd) of
   "advance" : (caller) % wait for "insert" message
               [ (discard, v) send (v) to caller ]
   "insert"  : (v)
               let hd = queue_cell(v,"nil")
               let tl = hd
               *[ case (cmd) of
                  "insert"  : (v)
                              let nt = queue_cell(v,"nil") % new tail
                              send ("link",nt) to tl
                              tl := nt
                  "advance" : (caller)
                              send ("get",self) to hd
                              [ case (cmd) of
                                "insert"   : (v)
                                             let nt = queue_cell(v,"nil")
                                             send ("link",nt) to tl
                                             tl := nt
                                             repeat
                                "hd reply" : (new_head,v)
                                             send (v) to caller
                                             hd := new_head
                              ]
                              if hd = "nil" then exit fi
                ]
 ]
```

Figure 2.8: Queue Object

2.4.3 Vectors

For the inner product programs of Figures 2.5 and 2.6, vectors were defined by linking together a bunch of element objects using let statements. A more general solution is to define a vector object as a function of its length, and then to instantiate and access the elements of the vector via indexed put and get messages. A vector object defined this way is shown in Figure 2.9. Initially the vector object waits for a message containing the limits of the vector: low and high. If low is less than high a vector object is created with the new reference value assigned to the local variable next. The new vector object is sent an initialize message with low incremented by one. The vector object then accepts "put" and "get" messages after entering the case description. Both message types contain a variable named index to compute the desired offset into the vector, starting with the first vector object. For both message types, the offset computation is performed by decrementing the index message variable for each vector object that the message passes through. When the value of index is decremented to 1, the receiving vector object performs the corresponding put or get operation.

For a vector of length N, if the indices sent to the vector are random, then the average number of messages sent per indexing operation is $\frac{N}{2}$. This number is excessive unless the vector can be used in a pipelined fashion, which effectively defeats the purpose of the random access mode. A second formulation for the vector object is shown in Figure 2.10. The elements of the vector are organized as a binary tree instead of a linear list. The maximum number of messages to access an element of the vector is $\log_2 N$. This worst case number is a tremendous improvement over the average case number for the linear list.

The interface to the vector object is unchanged from the linear version. Initially a message is sent to the vector object containing the lower and upper limits of the vector. The vector object calculates its index value and stores the result in the local variable me. If low is less than me, then a left sub-tree is created. Likewise a right sub-tree is created if high is greater than me.

After the initialization message has been processed, the vector object awaits "put" and "get" messages. Both message types perform the indexing operation by comparing the value of the index received to the value of me. If index equals me, the put or get operation is performed. Otherwise the "put" or "get" message is relayed to left if index is less than me, or is relayed to right if index is greater than me.

To reduce the access time further, the vector object could be implemented as a *custom* object. Custom objects use information about the runtime environment for the program that is outside the scope of Cantor. The vector object is a prime candidate for customization when there are large amounts of contiguous storage for objects. The interface to the vector object is identical to the interface used for the linear and tree versions. However, the indexing operation is reduced to a single send by replacing the linear or tree search with an address calculation into storage.

```
vector (content) ::
[ (low,high)
  let next = "nil"
  if low < high
     then next := vector(content)
          send (low+1,high) to next
  fi
  *[ case (cmd) of
        "put" : (index, val)
                if index = 1
                   then content := val
                   else send ("put", index - 1, val) to next
                fi
        "get" : (index, caller)
                if index = 1
                   then send (content) to caller
                   else send ("get",index - 1, caller) to next
                fi
   ]
]
```

Figure 2.9: Vector Object Organized as a Linear Linked List

```
vector (content) ::
[ (low,high)
  let me  = low + (high-low+1) / 2
  let left = "nil"
  let right = "nil"
  if low < me
     then left := vector(content)
          send (low, me-1) to left
  fi
  if high > me
     then right := vector(content)
          send (me+1, high) to right
  fi
  *[ case (cmd) of
     "put" : (index, val)
             if index < me
                then send ("put",index, val) to left
                else if index >  me
                        then send ("put",index, val) to right
                        else content := val
                     fi
             fi
     "get" : (index, caller)
             if index < me
                then send ("get",index,caller) to left
                else if index > me
                        then send("get",index,caller) to right
                        else send (content) to caller
                     fi
             fi
   ]
]
```

Figure 2.10: Vector Object Organized as a Binary Tree

2.4.4 Two-Dimensional Arrays

A straightforward method for generating two-dimensional arrays is to augment the vector object of Figure 2.9. The strategy is to consider an array of size M rows by N columns as a vector of length MN. Each array object would have two more persistent variables than the vector object. One variable would provide the next row link for each element of the array and the other would be a back link for propagating messages right to left.

Messages would flow in both directions along the doubly linked chain of objects. Messages would travel left to right as before to link the column elements of one row and then then thread contiguous rows together in a serpentine fashion. Messages flowing back along the chain, i.e. right to left, would be used to set the next row link for each element by propagating a reference value for an array object from right to left N elements. Excluding the first row, a message sent back N places will line up with the array object that is in the row directly above the originating sender.

There are two drawbacks to this approach. The first is that the array is created in a completely sequential fashion. The doubly linked chain of MN objects is created in a linear fashion and then the next row links are set by propagating messages backwards along the chain. The second drawback is that, excluding the first and last row, each array object originates one back flowing message, but also forwards $N - 1$ more messages, the total number of messages is therefore $O(MN^2)$.

A more concurrent solution that requires fewer messages is shown in Figure 2.11. The overall strategy is to progressively build the array along a diagonal that starts with the upper left hand corner of the array ($m = 1, n = 1$). Excluding the first row and column, and last row and column, each object waits for two messages, a "diag" message from the array object that is one row above and one column to the right of the receiving object $(m - 1, n + 1)$, and an "up" message from the array object in the row directly beneath the receiving object. Each object will receive exactly one "diag" and one "up" message but their arrival order is unknown. Thus if a "diag" message arrives first, the object will wait for a "up" message and likewise if an "up" message arrives first.

For each pair of messages received, the object will create a new array object for the next column and send an "up" message to the object which is one row above and one column over from the current object. The array object uses the content of the "up" message to set the next_row persistent variable and uses the content of the "diag" message as the destination for the "up" message.

After the next_row persistent variable is set via an "up" message and the next_col persistent variable is set by creating a new array object, a "diag" message is sent to the array object of next_row with a one row up and one column over value of next_col. Care has to be exercised for "edges" of the array because array objects in the first row and last column will never receive "diag" messages, and array objects in the last row will never receive "up" messages.

To detect when the array has been completely built, for every row completed an empty message is sent to caller. Thus caller is expected to keep a count of the number of rows in the array and also a count of the number of replies it has received from the array. When the number of replies received equals n, caller may send the array new messages. For this example, only two operations have been defined on the array, "put i j" and "get i j". These two message types are similar to the messages types defined for the vector object. For the put and get messages to reach their destination, they are first sent along the first column, decrementing the message variable i until the destination row is reached and are then sent

along the destination row, decrementing the message variable j until the destination array element is reached.

Like the linear vector object, if the indices sent to the array are random, then the average number of messages sent per index operation is $\frac{M+N}{2}$. This number can be reduced by building the matrix out of the tree vector objects. For this scheme, the matrix is placed into row or column major form. For the row major form, the matrix contains a binary tree of M vector objects. Each vector object has reference to its left and right sub-tree, and also a reference to the column tree for that object. Once the tree has been traversed to find the correct row index, the column tree is traversed to find the correct column index. The column major form is identical except that the roles of row and column are interchanged. The maximum number of sends for this scheme for any pair of indices is $\log_2 MN$.

```
array (max_m, max_n, next_col, next_row, content) ::
*[[ case (cmd) of
  "diag" : (m, n, up_and_over, caller)
           if n < max_n
              then next_col := array(max_m, max_n, "nil", "nil", 0.0)
                   if m > 1
                      then send ("up", m-1, n+1, next_col, caller)
                             to up_and_over
                   fi
              else send () to caller
           fi
           if m < max_m
              then if n > 1
                      then [ (cmd, m, n, down_link, caller)
                             next_row := down_link
                           ]
                      else next_row := array(max_m, max_n, "nil", "nil", 0)
                   fi
                   if n < max_n
                      then send ("diag",m+1,n,next_col,caller) to next_row
                   fi
           fi
  "up"   : (m, n, down_link, caller)
           next_row := down_link
           if n < max_n
              then next_col := array (max_m, max_n, "nil", "nil", 0)
                   send ("diag", m+1, n, next_col,caller) to next_row
              else send () to caller
           fi
           if m > 1 and n < max_n
              then [ (cmd, m, n, up_and_over, caller)
                     send ("up",m-1,n+1,next_col,caller) to up_and_over
                   ]
              else if m = 1 and n = max_n then send () to caller fi
           fi
 ]
 [ case (cmd) of
"put i j" : (i, j, val)
            if i > 1
               then send ("set i j", i-1, j, val) to next_row
               else if j > 1 then send ("put i j", i, j-1, val) to next_col
                             else content := val
                    fi
            fi
"get i j" : (i, j, caller)
            if i > 1
               then send ("get i j", i-1, j, caller) to next_row
               else if j > 1 then send ("get i j", i, j-1, caller) to next_col
                             else send (content) to caller
                    fi
            fi
 ] ]
```

Figure 2.11: Two-Dimensional Array Object

Chapter 3

Three Programming Examples

Writing concurrent programs using the objects of Cantor usually requires a considerable amount of forethought about the *concurrent formulation* for the program. The objective for the concurrent formulation phase of program writing is to determine how the various concurrent objects are to be organized and how the various components for the "state" of the computation are to be partitioned among the objects.

The process of developing good concurrent formulations, whether by intuition, technique, or a combination of both, is fundamental to writing programs. To serve as a starting point for writing useful programs in Cantor, this section examines in detail three medium size programs. The discussions that follow place emphasis upon the concurrent formulations, trusting that the reader has a working knowledge of the individual mechanisms of Cantor. The historical origin for these examples is the XCPL report [3]; their inclusion here is a virtual wholesale rip-off from the XCPL report.

3.1 Generating Prime Numbers by Sieving

The programming task is to generate all the prime numbers up to a predefined limit by constructing a "sieve" that filters out all the non-prime numbers. The technique is devoid of tantalizing number-theoretic tricks but uses a simple and elegant concurrent formulation. The program is partitioned into two parts: the sieve and the number generator that feeds the sieve. The requirements of the number generator are that it must emit a stream of integers such that the stream contains all the prime numbers and that the integers are emitted in strictly increasing order. The simplest such stream would be the natural numbers starting with 2.

The sieve can be described by a linear chain of objects with each object containing a single prime number. Each sieve object receives potential primes numbers to test for divisibility. If the test number divides evenly, it is discarded; otherwise the test number is relatively prime and is sent to the next sieve element. Whenever a test number reaches the end of the sieve, the number must be a prime and is made into a new sieve object at the end of the chain.

The number generator could be a simple counter and emit the natural numbers starting with 2. Naturally this would be wasteful, for all even numbers except for 2 are not prime. A first improvement would be to omit all multiples of two, i.e. emit only the odd numbers. Further improvements would be to omit multiples of 2,3,5,7,11, etc.. A number generator of this type is called a "wheel" [6]. The wheel is composed of an "addendum" and "spokes". The addendum is the product of the prime numbers from 2 up to some finite limit. The

spokes are the integers relatively prime to the addendum. The wheel operates by maintaining an accumulator that is a multiple of the addendum. The accumulator is added to each spoke and then sent to the sieve for primality testing. After each spoke has sent a number to the sieve, the accumulator is advanced to the next multiple of the addendum.

For example, consider the addendum of: $1 \cdot 2 \cdot 3 = 6$. This addendum would result in a two spoke wheel of 1 and 5, and would generate the sequence:

$$0+1, 0+5, 6+1, 6+5, 12+1, 12+5, 18+1, 18+5, ... = 1, 5, 7, 11, 13, 17, 19, 23...$$

Every integer in this sequence is relatively prime to 2 and 3, and the first non-prime generated is 25. The number of spokes for a wheel is determined by calculating Euler's Function for the addendum.

3.1.1 Wheel Driven Primes Sieve

The program of Figure 3.1 uses the combination of a predetermined object structure and a computed object structure. The wheel is the predetermined object structure, consisting of 8 spokes initialized with the values of 1,7,11,13,17,19,23, and 29. The addendum for the wheel has the value: $2 \cdot 3 \cdot 5 = 30$. The computed object structure is the sieve. A snapshot of the object graph during the execution of this program is shown in Figure 3.2. In this graph, circles represent objects and the edges represent the references between objects. The program of Figure 3.1 requires only three types of objects.

The sieve objects make up the linear chain of objects that comprise the sieve. The wheel is comprised of 8 spoke objects and a single idler object. The 8 spoke objects and the idler object are linked together inside of the main object as a ring. Initially a single message is inserted into the ring at spoke3. Thereafter a message will circulate through the 9 objects that comprise the wheel. For each complete revolution each spoke object will insert a test number into the sieve. The idler object will then advance the current value of the accumulator by the value of the addendum. This cycling action will continue until the accumulator reaches 10,000 at which point the wheel is shut down followed by the sieve emptying out.

The sieve object performs as described previously. Each sieve object independently sends prime number messages to console, so prime numbers may arrive at the external display object in non-ascending order.

By making some basic assumptions about the *environment* of the objects, the concurrent behavior of the primes program can be measured quantitatively as a function of time. For each time step or "sweep", every object that has one or more messages enqueued is considered "active". The set of active objects establishes the number of messages that can be processed concurrently. The size of the active object set is the *concurrency index* (CI) for the sweep. The assumption to be made is that all active objects process a single message in the same amount of time. This time interval is the *sweep*.

As a first approximation, the assumption that all active objects process messages within the same time quantum is acceptable because iteration is not allowed inside of the objects and because the "bulk" of the concurrent objects are created from the same object definition. Differences in execution time per message received are limited to the number of different object definitions and data dependencies.

A second assumption is that message delivery is instantaneous.

From these two assumptions, the rules for running a program are to allow each of the active objects to process a single message, tally the total number of objects and the number

```
spoke (v,hub,next) ::
*[ (k)
   send (v+k) to hub
   send (k) to next
]

sieve (v. next. output) ::
*[ (p)
   if (p mod v) ⟨⟩ 0)
     then if next = nil
            then send (p) to output
                 next := sieve(p, "nil", output)
            else send (p) to next
          fi
   fi
]

idler (delta, next) :: *[ (k) if (k ⟨ 10000) then send (delta+k) to next fi ]


[ (console)
  let hub = sieve (7,"nil",console)
  let spoke1 = spoke( 1,hub,spoke2)   let spoke2 = spoke( 7,hub,spoke3)
  let spoke3 = spoke(11,hub,spoke4)   let spoke4 = spoke(13,hub,spoke5)
  let spoke5 = spoke(17,hub,spoke6)   let spoke6 = spoke(19,hub,spoke7)
  let spoke7 = spoke(23,hub,spoke8)   let spoke8 = spoke(29,hub,idler1)
  let idler1 = idler(30,spoke1)
  send (0) to spoke3
]
```

Figure 3.1: Wheel Driven Prime Sieve

of active objects, and then iterate. Figures 3.3 and 3.4 show the number of objects and concurrency index as a function of the sweep count for the wheel driven prime sieve. The program used to make the two graphs is identical to the program of Figure 3.1 except that the message send to output in the sieve object was removed. The number of sweeps necessary to compute all primes less than 10,000 is 4,241. This is the absolute minimum, assuming that every opportunity for concurrency is exploited. The number of objects grows at a steady, monotonically increasing rate, reflecting the growth of the sieve.

The concurrency index also grows steadily since concurrent behavior is achieved by the stages of the sieve working concurrently. The concurrency index continues to increase until the number generator that is input to the sieve is shut down. The sieve then begins to empty out with the concurrency index monotonically decreasing.

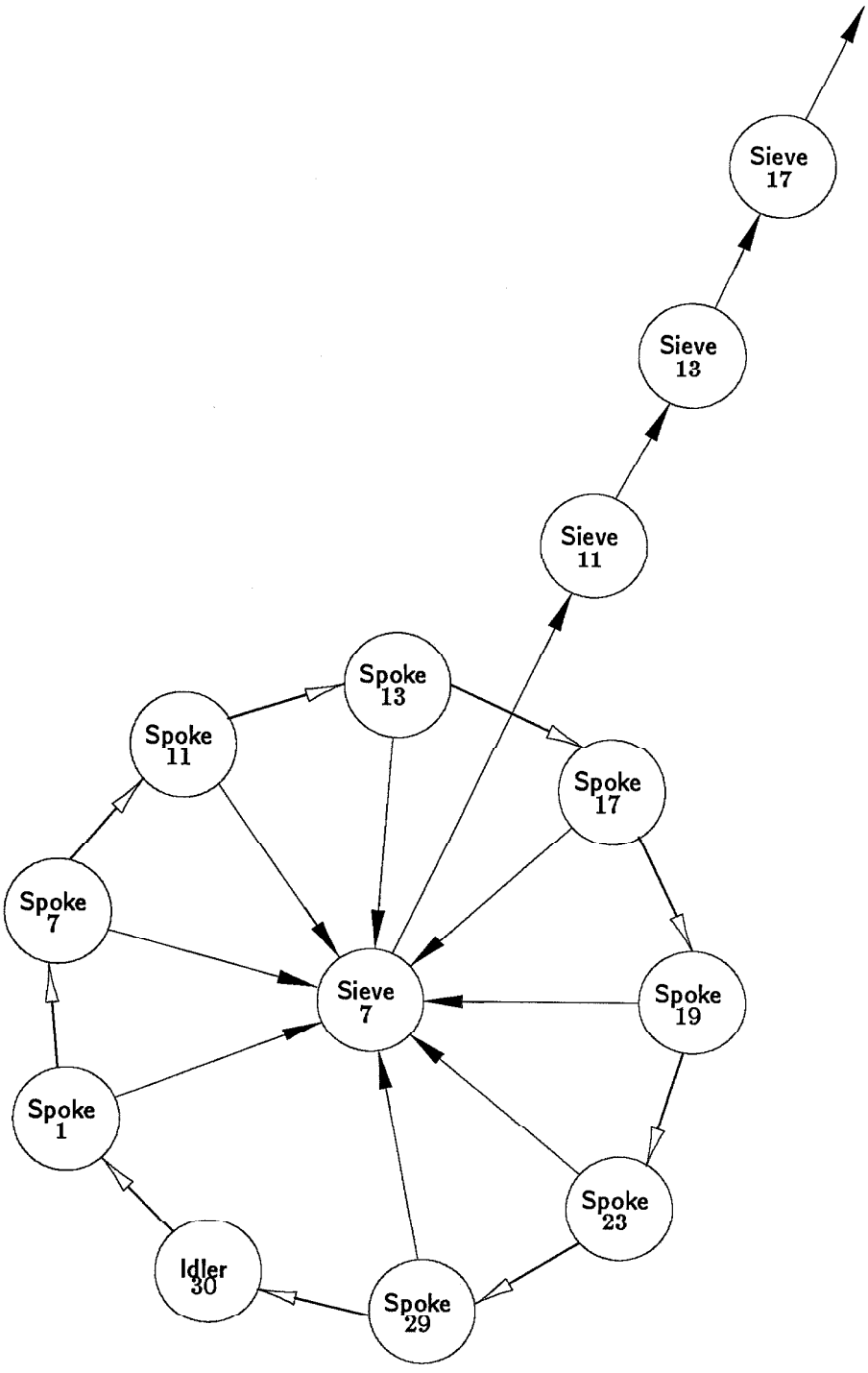

Figure 3.2: Object Graph for Prime's Sieve

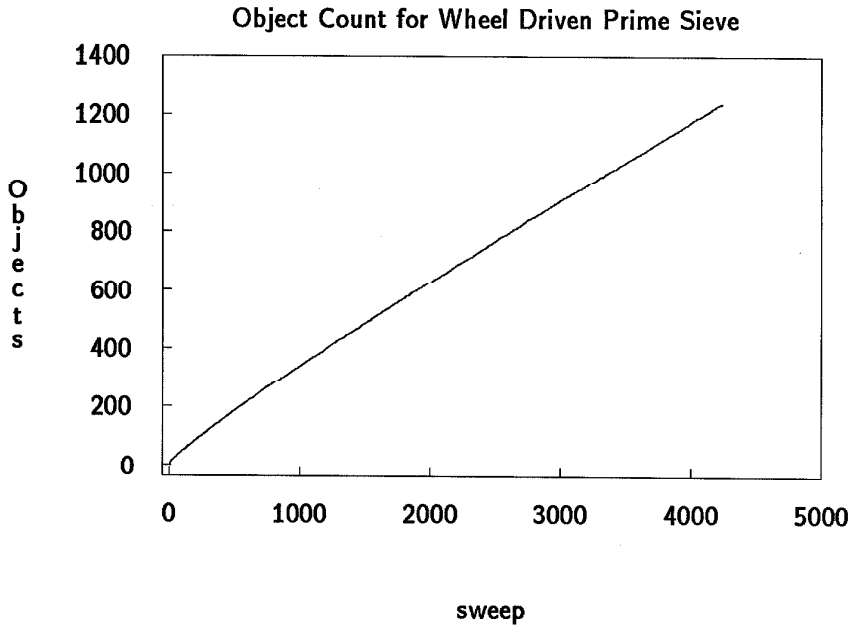

Figure 3.3: Object Count for Wheel Driven Prime Sieve

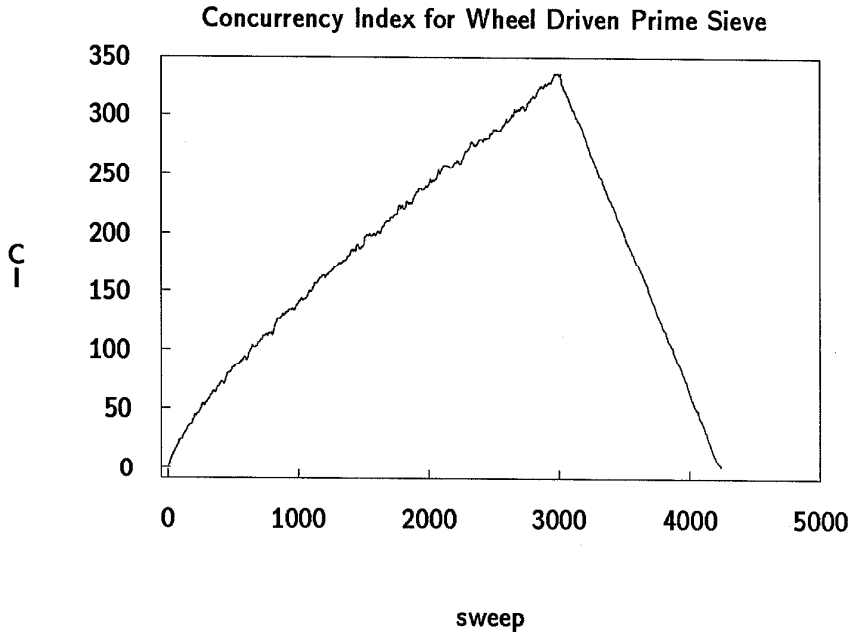

Figure 3.4: Concurrency Index for Wheel Driven Prime Sieve

3.1.2 Demand Driven Primes Sieve

A major problem with the program of Figure 3.1 is that the production of messages flowing into the sieve is not regulated to the consumption rate of the sieve. Therefore a message bottleneck may form between the spokes of the wheel and hub. A solution to the bottleneck between the sieve and the wheel that does not change the basic character of the algorithm is to alter the message production so that the "supply" of test numbers never exceeds the "demand" for test numbers. Stated slightly differently, prime numbers will be "pulled" out from the end of the sieve instead of having test numbers "pushed" into the first stage of the sieve.

A program that uses the new strategy is shown in Figure 3.5. Each sieve object employs the case construct to distinguish between two types of incoming messages, "request" and "answer". The sieve objects are created containing a prime number p and are initially waiting for a number N to test for primality. For each "answer" message received, the sieve object will immediately request another test number and then test the number received for primality. If the number is relatively prime, the number is sent to next which is initially set to the value of the external object console. A new sieve object is then created and assigned to the next acquaintance variable.

After the next acquaintance variable contains a reference value to a new sieve object, the original sieve object can be in one of two modes, either the object contains a test number that is relatively prime and is waiting for a "request" message from the next sieve element, or has received a "request" message from the next sieve element and is waiting for a test number from the previous stage of the sieve. Notice that the sieve object sends an initial "request" message to self to compensate for the missing "request" message from the new sieve object.

The overall strategy for the sieve objects is to internally "buffer" one number that is relatively prime to the number stored inside the sieve object. When a "request" message is received, the sieve object will repeatedly send "request" messages to the previous sieve element until a relatively prime number is received. The number is then sent onto the next stage of the sieve. If the "answer" message is received first and the number N received is relatively prime, the object waits for the "request" message and then sends the saved value of N to the next stage of the sieve.

The first sieve object is connected to a new object called axle which serves as the interface between the wheel and the sieve. Each "request" message received by the axle object will cause the wheel to advance one notch. Thus the speed at which the wheel "rotates" is directly controlled by how fast the first sieve object can accept test numbers, and indirectly by the rate at which prime numbers are stored into new sieve objects at the end of the sieve.

The plots for the object count and concurrency index are shown in Figures 3.6 and 3.7. Like the plots for the wheel driven version, the message send to console inside of sieve was removed. The number of sweeps to completion is 12,269, which is about a factor of 3 greater than the wheel driven version. The overall shape of the two object graphs and concurrency index graphs are the same. The concurrency index graph for the demand driven primes sieve is "bumpier" than the wheel driven version. The factor of 3 difference in number of sweeps and the bumpiness is due to the handshaking between the stages of the sieve. For the wheel driven version, each transmission of an integer between sieve objects requires 1 message to be processed. For the demand driven version, 3 messages are processed: request integer, answer integer, and accept reply.

```
sieve (next, back, p) ::
*[ (cmd, N)
   send ("request") to back
   if N mod p = 0
      then send ("request") to back
      else send (N) to next
           next := sieve(next, self, N)
           send ("request") to self
           *[ case (cmd) of
              "request" : () *[ (cmd, N)
                                send ("request") to back
                                if N mod p <> 0
                                   then send ("answer", N) to next
                                   exit
                                fi ]
                "answer" : (N)
                           if N mod p = 0
                              then send ("request") to back
                              else let N_save = N
                                   [ (cmd) send ("answer", N_save) to next ]
                                   send ("request") to back
                           fi ]
   fi ]

axle (next_spoke, start, acc) ::
*[ (discard)
   send (acc) to next_spoke
   [ (v, k, ns) send ("answer", v + k) to start
                acc := k
                next_spoke := ns ] ]

spoke (v, hub, next) :: *[ (k) send (v, k, next) to hub ]

idler (delta, next) ::
*[ (k) if (k < 10000) then send (delta+k) to next fi ]

[ (console)
  let hub    = axle(spoke3, sieve(console, hub, 7), 0)
  let spoke1 = spoke( 1,hub,spoke2)   let spoke2 = spoke( 7,hub,spoke3)
  let spoke3 = spoke(11,hub,spoke4)   let spoke4 = spoke(13,hub,spoke5)
  let spoke5 = spoke(17,hub,spoke6)   let spoke6 = spoke(19,hub,spoke7)
  let spoke7 = spoke(23,hub,spoke8)   let spoke8 = spoke(29,hub,idler1)
  let idler1 = idler(30,spoke1)
  send ("request") to hub ]
```

Figure 3.5: Demand Driven Prime Sieve

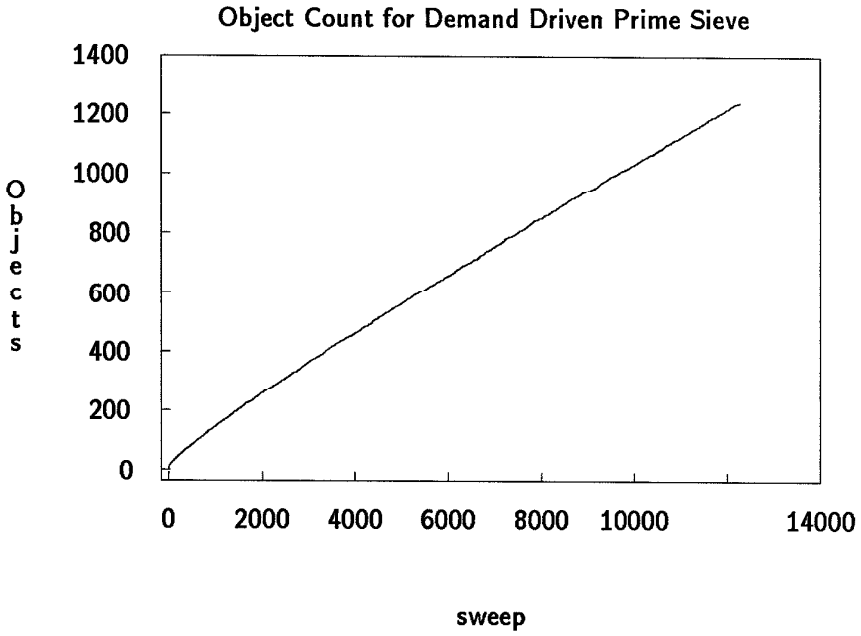

Figure 3.6: Object Count for Demand Driven Prime Sieve

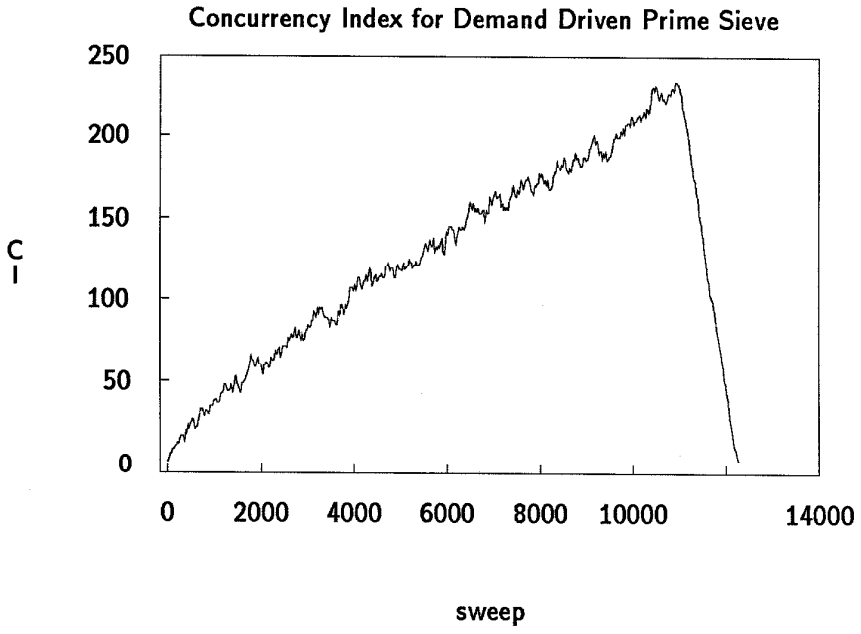

Figure 3.7: Concurrency Index for Demand Driven Prime Sieve

3.2 The Eight Queens Problem

The Eight Queens problem is one of the showcase problems often cited in the literature of computer programming. The task is to place eight queen pieces on an 8x8 chessboard so that no queen is in jeopardy of capture. The attack rule for the queen chess piece is that any piece lying along the same row, column, or either diagonal may be captured. The paradigm of "divide and conquer" is used to conduct a breadth-first search of the solution space.

A solution that uses only a precomputed object structure is shown in Figure 3.8. Each **queen** object represents one column of the chessboard. Since all solutions will have exactly one queeen per column and row, the program could also have been organized as one object per row. The **queen** objects are doubly linked together inside of the **main** object as a chain of eight objects. Each **queen** object is assigned a column number as an acquaintance variable. Both ends of the chain are terminated by a single instance of the **border** object.

A second message variable, called **chessboard**, has been included as part of the message list for the **main** object. The **queen** objects are programmed to send their results to this second external object instead of to **console**. The intent here is that the **chessboard** variable is a reference value for an external graphic display, suitable for displaying both the chessboard and the chess pieces in a visually appealing fashion.

A first solution is found by sending the **queen** object of column one a "move" message with destination row number equal to 1. When the **queen** object of column one accepts the "move" message, it will assign its acquaintance variable **row** to the value of the message variable **new_row**. The **queen** object then checks to see if its column number is the final column number. If the column number is less than eight, an "evade" message is sent to self to start searching for a safe row number for the **queen** object in the column to the right of the current **queen** object.

When the **queen** object accepts the "evade" message, it will check whether the test row number is on the board. If the test row number is eight or less, a "capture" message is sent to self to test whether the test row and column numbers for the **queen** object of the next column can be captured by the current **queen** object. If the projected coordinates are safe, then the "capture" message is sent to the **queen** object in the column to the left of the current object. This action will continue recursively until either the test row and column numbers cause a capture to occur, or the "capture" message reaches the **border** object.

If the test position causes a capture to occur, the **queen** object that initiated the search for the new column is sent an "evade" message with the test row number incremented. If the "capture" message reaches the **border** object, then the test row and column numbers are safe from capture and the **queen** object corresponding to the test column number is sent a "move" message with the test row number as the content of the message.

Whenever an "evade" message forces the test row number for the new column off the board, the current configuration of queens will not produce a solution. The search then needs to be retracted to searching for another safe position for the current column, instead of the next column. This "backtracking" is accomplished by having the current **queen** object send the **queen** object in the column to the left an "evade" message with a new test row number for the current **queen**. The search for a safe position is then backed off one column, and will continue backing off all the way back to the **border** object. If the **border** object ever receives an "evade" message, then no solution exists.

If, however, a safe position is found for **queen** object of column eight, then a complete solution has been found. The **queen** object of column eight will then send itself an "ack"

```
border (output) ::
*[case (cmd) of
"capture" : (test_col,test_row, tester, next)
            send ("move", test_row) to next
  "evade" : () send ("no solution") to output
    "ack" : () send () to output
            exit
 ]

queen (col, row , left, right, output) ::
*[case (cmd) of
  "ack"   : () send (col,row) to output
            send ("ack") to left
            exit
  "move"  : (new_row)  row := new_row
                       if col = 8
                          then send ("ack") to self
                          else send ("evade", 1) to self
                       fi
  "evade" : (test_row)
            if test_row > 8
               then send ("evade",row+1) to left
               else send ("capture",col+1,test_row,self,right) to self
            fi
"capture" : (test_col, test_row, tester, next)
            if (row = test_row) or
               ((test_col - col) = (abs (test_row - row)))
               then send ("evade",test_row+1) to tester
               else send ("capture",test_col,test_row,tester,next) to left
            fi
]

[ (console, chessboard)
  let bb = border (chessboard)
  let q1 = queen (1,1,bb,q2,chessboard)
  let q2 = queen (2,1,q1,q3,chessboard)
  let q3 = queen (3,1,q2,q4,chessboard)
  let q4 = queen (4,1,q3,q5,chessboard)
  let q5 = queen (5,1,q4,q6,chessboard)
  let q6 = queen (6,1,q5,q7,chessboard)
  let q7 = queen (7,1,q6,q8,chessboard)
  let q8 = queen (8,1,q7,bb,chessboard)
  send ("move",1) to q1
]
```

Figure 3.8: Eight Queens Program Organized by Columns

message. The response of a queen object to an "ack" message is to send the internal row and column numbers to the reference value contained in output, which is always set to the value of chessboard. The "ack" message is then forwarded to the queen object in the column to the left and will propagate from right to left until an "ack" message reaches the border object.

An interesting property of this program is that from the first solution all subsequent solutions can be found. These additional solutions are extracted simply by sending column eight "evade" messages until the border objects sends a "no solution" message.

An unfortunate property of this program is that it is sequential. A small improvement would be to change the strategy of the capture check so that the new coordinates for a queen's position are sent to all of the columns to the left of the current queen object concurrently, instead of performing the capture check one column at a time. The larger problem however stems from the conservative nature regarding the processing of the "evade" messages. Only a single new column position is investigated rather than checking all of the possible new positions.

The program shown in Figure 3.9 performs a concurrent search for the set of possibly safe new row and column numbers. The doubly linked list of main is replaced by a tree whose depth and breadth can only be determined by running the program. The tree is constructed by sending a "start" message to the top object which will create eight queen objects, one for each row of the chessboard with column number always initialized to one. The queen objects are sent "check" messages which replace the "evade" messages. When a queen object receives a "check" message, it will check whether the eighth column has been reached. If so a "print" message is sent up the tree in the same fashion as was used for the sequential solution. Otherwise eight "capture" messages are sent to self to perform the concurrent search for the next column

The response to a "capture" message has been modified so that when a capture occurs, the "capture" message is discarded, thus ending the search. If the "capture" message reaches the top, then the queen object that originated the search is sent an "add queen" message. When the "add queen" message is received, a new queen object is generated and the entire process continues in a recursive fashion.

For this concurrent solution no backtracking is necessary. New positions are investigated concurrently and whenever a capture occurs, the unsafe coordinates are discarded. If the "capture" messages reaches top, then a new leaf node is added to the tree. Thus the strategy has shifted from using sequential search with backtracking to a concurrent breadth-first search.

There are two problems with this solution. First, detection of when the program has terminated is not self-evident. For the sequential program, a single solution was expected and when that solution arrived, the program was known to have completed. For the concurrent version, the number of expected solutions is unknown, and determining when the program has become quiescent cannot be determined solely from the program output. This problem is not too serious, for termination can be determined by the Cantor runtime system.

A more serious problem with this solution is that the individual coordinates for queen game pieces are sent to chessboard in an interleaved fashion.

```
top (output) ::
*[ case (cmd) of
    "start" : (i) send ("check", 1) to queen (self, i, 1, output)
                  if i < 8 then send ("start",i+1) to self fi

  "capture" : (test_col, test_row, checker)
              send ("add queen", test_row) to checker

  "print"   : () send () to output
 ]

queen (parent, row, col, output) ::
*[ case (cmd) of
  "print" : () send (row,col) to output
               send ("print") to parent

"add queen" : (ok_row) send ("check",1) to queen(self,ok_row,col+1, output)

  "check" : (test_row)
            if col < 8 then send ("capture", col+1, test_row, self) to self
                       else send("print") to self
            fi
            if test_row < 8 then send ("check", test_row+1) to self fi

"capture" : (test_col, test_row, checker)
            if (row = test_row) or
               ((test_col - col) = (abs (test_row - row)))
               then repeat
               else send ("capture",test_col, test_row, checker) to parent
            fi
]

[ (console, chessboard) send ("start",1) to top(console) ]
```

Figure 3.9: Concurrent Eight Queens Program

```
top (linker) ::
*[ case (cmd) of
    "start" : (i) send ("check", 1) to queen (self, i, 1)
                  if i < 8 then send ("start",i+1) to self fi

  "capture" : (test_col, test_row, checker)
              send ("add queen", test_row) to checker

  "print"   : (first, last) send (first,last) to linker
 ]

spooler (console,i) ::
*[ (row, col, caller) send (col,row) to console
                      i := (i + 1) mod 8
                      if (i = 0) then send () to console fi
                      send ("advance",self) to caller
 ]

link (output) ::
[ (first, tail) send ("print",output) to first
                *[ (first , last) send ("enlink",first,output) to tail
                                  tail := last
                 ]
]

chain (next, row, col) ::
*[ case (cmd) of
"enlink"  : (first, printer) next := first
"print"   : (printer) send (row,col,self) to printer
"advance" : (caller)
            if next = "nil"
               then [ (cmd,first,printer) send ("print",printer) to first ]
               else send ("print", caller) to next
            fi exit
 ]
```

Figure 3.10: Output Control for Concurrent Eight Queens Program

```
queen (parent, row, col) ::
*[ case (cmd) of
  "print" : (down, bottom)
            send ("print",chain(down.row.col).bottom) to parent

"add queen" : (ok_row)    send ("check",1) to queen(self,ok_row,col+1)

  "check" : (test_row)
            if col < 8 then send ("capture", col+1, test_row, self) to self
                       else let b = chain("nil",row,col)
                            send("print",b,b) to parent
                            exit
            fi
            if test_row < 8 then send ("check", test_row+1) to self fi

"capture" :  (test_col, test_row, checker)
             if (row = test_row) or
        ((test_col - col) = (abs (test_row - row)))
                then repeat
                else send ("capture",test_col, test_row, checker) to parent
             fi
]

[ (console, chessboard)
  send ("start",1) to top (link(spooler(chessboard,0)))
]
```

Figure 3.11: Concurrent Eight Queens Program with Output Control

Clearly a mixture of the two strategies is desired, i.e. a concurrent search followed by a meaningful serialization of the results as they are sent to chessboard. One possible composition of the two strategies is shown in Figures 3.10 and 3.11. The search for solutions is unchanged; however, the printing of a solution is handled substantially differently. Three new objects definitions are introduced in Figure 3.10, chain, spool and link. The chain object constructs a linear list of eight queen coordinates to represent a single solution. The spool object feeds the lists of solutions to the chessboard display. The link object serializes the lists as they exit the root of the tree by linking them together into longer lists.

The construction of the linear lists replaces the sending of the solution coordinate pairs directly to chessboard. When a "check" message is received and the column number is eight, a print list is started by creating a chain object and sending a "print" message up the tree as before. As the print list grows from the leaf node of the tree to the root node, both the head and tail of the list are passed along as part of the "print" message. When the "print" message reaches the top object, it is sent to the link object, which immediately prints the first list it has received by sending the first element of the print list a "print" message with a reference to the spool object.

When the chain object receives a "print" message, the row and column numbers internal to the chain object are sent to the spool object. The spool object will send the coordinates to the chessboard and then send an "advance" message back to the originating chain object. To separate the display of the solutions, the spool object is coded to send an empty message to the chessboard after the eight coordinates of a complete solution have been sent to chessboard.

When the chain object receives an "advance" message, one of two conditions may hold, either the print list has been emptied, or there is another chain object following the current chain object. If the end of the list has been reached, the object waits for a new "enlink" message from the link object and then restarts the "print" and "advance" message cycle. Otherwise the "print" and "advance" message cycle starts immediately on the following chain object.

The relative speeds between the chain, spool, link, and queen objects is therefore inconsequential. If the solutions are sent to the chessboard much faster than they are produced, then the chain object which was the last to be displayed will wait for an "enlink" message. If the answers are produced faster than they can be sent to the chessboard, then they are spliced together to form a list of solutions. The behavior of the "print" and "advance" message cycle is comparable to the protocols used in self-timed system design.

The plots for the object count and concurrency index are shown in Figures 3.12 and 3.13. The source program for these plots is the same as the program in Figure 3.10. The output control of Figure 3.11 was simplified only to count and print the number of solutions. The reason for this change was that the printing phase of the solutions is completely sequential.

The concurrency index for the concurrent queens program reaches a peak value of 38. Intuitively the program should have significantly more concurrency since the program is building an 8-way tree of depth eight. Figure 3.14 shows the *message load* (ML), which is defined as the number of outstanding messages divided by the number of active objects per sweep. This quantity should always be greater than or equal to one. For the two versions of the primes sieve, the number was always one and hence the message load graphs were omitted. The concurrent eight queens program however has a high message load, with a peak value of about 20 after excluding the final sweeps where messages sent to the console object form a bottleneck.

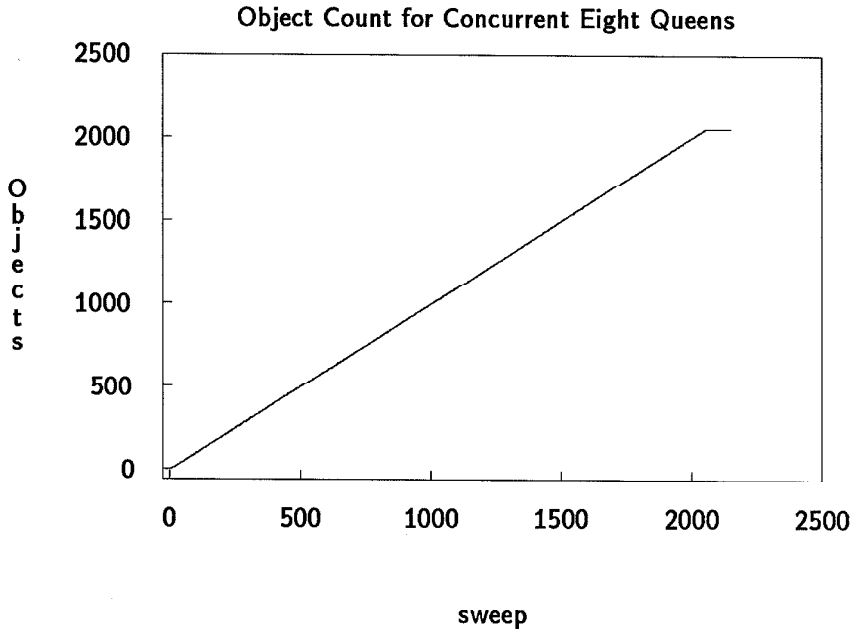

Figure 3.12: Object Count for Concurrent Eight Queens

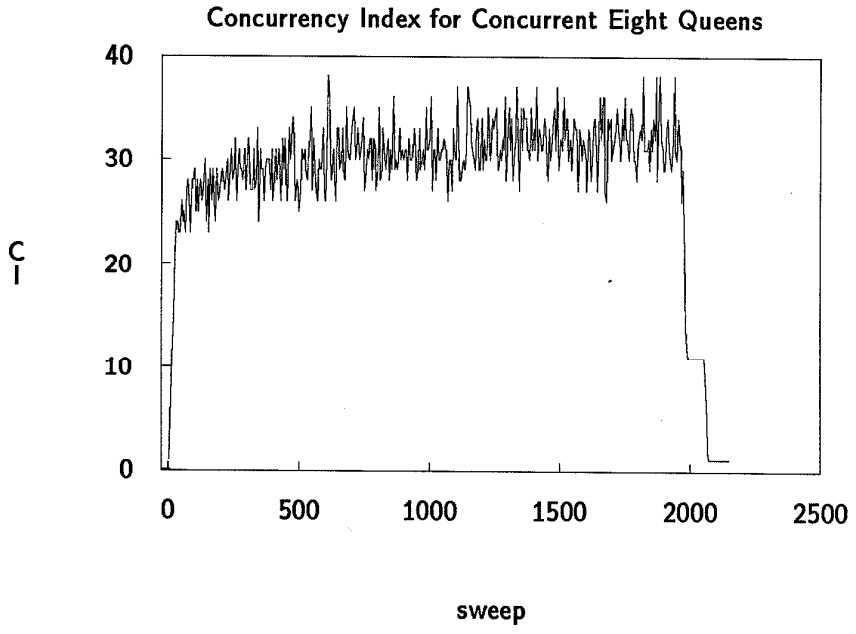

Figure 3.13: Concurrency Index for Concurrent Eight Queens

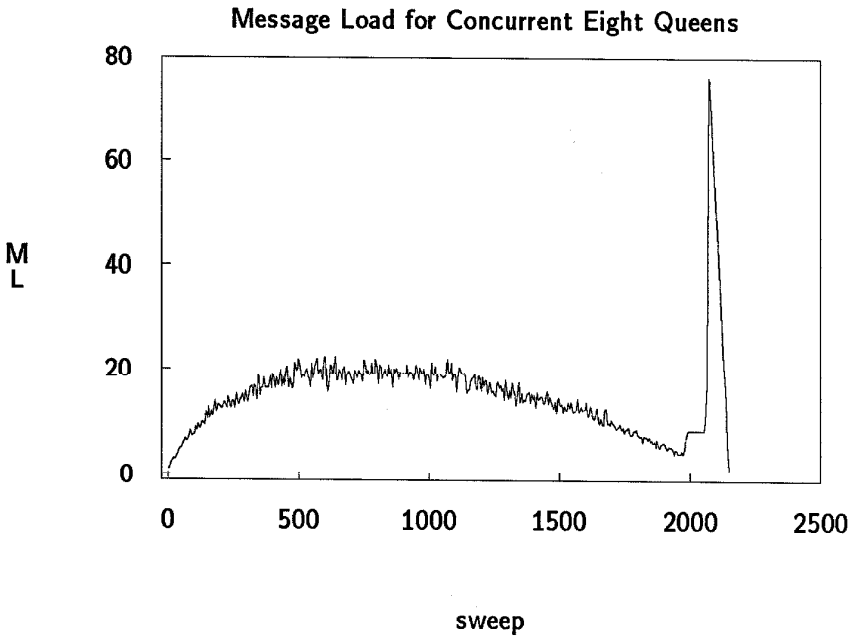

Figure 3.14: Message Load for Concurrent Eight Queens

High message load values indicate that there is a bottleneck somewhere in the program. Careful examination of the object definition for **queen** shows that a bottleneck in message flow could occur in the "**add queen**" case where a new queen is added by creating a new **queen** object whose parent link is set to self. This use of self minimizes the number of queen objects but requires all of the sibling **queen** objects to send "**capture**" messages to the single parent. If self was replaced by **queen(parent, self, col)**, then each new **queen** object would have a separate parent **queen** object which to send "**capture**" messages.

The revised program for the concurrent eight queens is shown in Figure 3.15. The plots for object count, concurrency index, and message load are shown in Figures 3.16 through 3.18.

The revised program generates about twice as many objects, but the concurrency index is about a factor of 7 greater and the number of sweeps to completion is about a factor 7 smaller. The area under the two concurrency index curves is about the same, indicating that the total number of messages that have to be processed for the two programs is also approximately equal. More concurrency is exploited in the revised program by parceling out the work load among more objects. The message load graph of Figure 3.18 for the revised program supports this conclusion.

```
counter(output, n) :: *[ () send (n) to output
                           n := n + 1
                       ]

top (output) ::
*[ case (cmd) of
    "start" : (i) send ("check", 1) to queen (top(output), i, 1, output)
                  if i < 8 then send ("start",i+1) to self fi

  "capture" : (test_col, test_row, checker)
              send ("add queen", test_row) to checker
 ]

queen (parent, row, col, output) ::
*[ case (cmd) of
"add queen" : (ok_row)
              send ("check",1) to queen(queen(parent, row, col, output),
                                        ok_row,col+1, output)

  "check" : (test_row)
            if col < 8
               then send ("capture", col+1, test_row, self) to self
                    if test_row < 8
                       then send ("check", test_row+1) to self
                    fi
               else send () to output
                    exit
            fi

"capture" :  (test_col, test_row, checker)
             if (row = test_row) or
                ((test_col - col) = (abs (test_row - row)))
                then repeat
                else send ("capture",test_col, test_row, checker) to parent
             fi
]

[ (console) send ("start",1) to top(counter(console,1),1) ]
```

Figure 3.15: Revised Concurrent Eight Queens Program

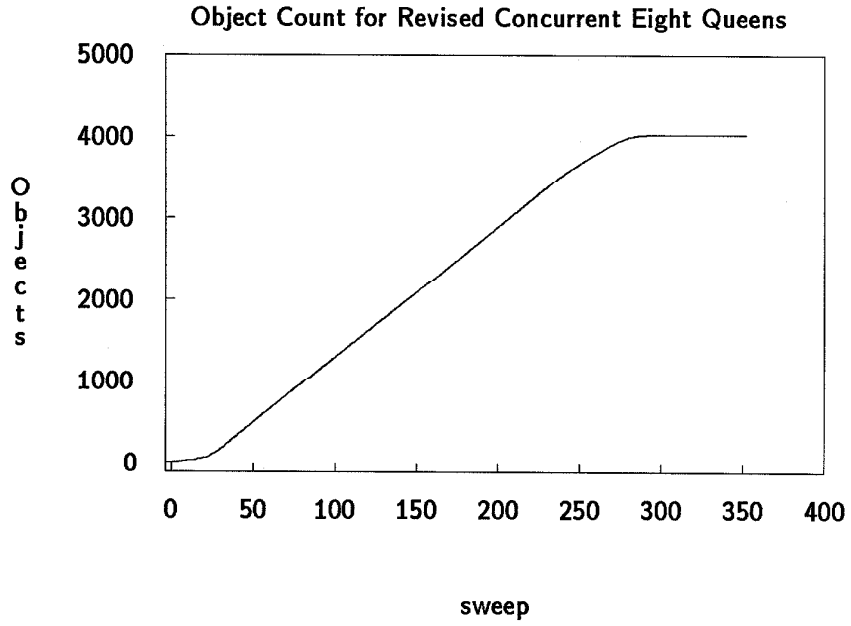

Figure 3.16: Object Count for Revised Concurrent Eight Queens

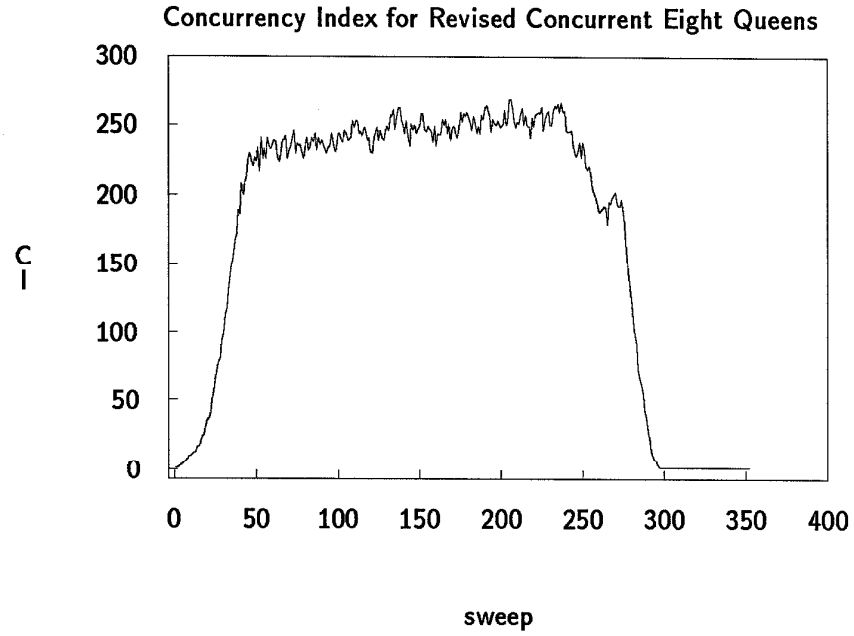

Figure 3.17: Concurrency Index for Revised Concurrent Eight Queens

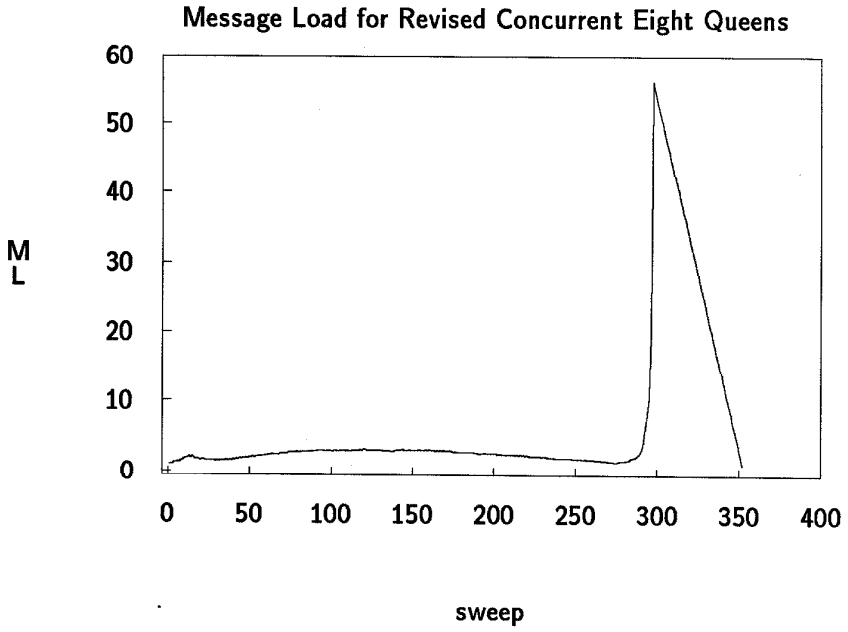

Figure 3.18: Message Load for Revised Concurrent Eight Queens

3.3 Gaussian Elimination

Numerical analysis problems present a special challenge when programmed in Cantor. Programs used in numerical analysis are often formulated in terms of matrices and described as sequences of actions performed on the indexed elements of the matrices. Such a representation provides a reasonable compromise between the domain of mathematical objects used by numerical analysts and the representation of matrices as a segment of contiguous storage in sequential computers. Cantor does not provide for contiguous arrays of storage elements as one of the basic types of values. Instead, matrices are built out of Cantor objects as was described in Chapter 2 for the vector and array objects.

A direct translation of a sequential program's manipulation of a matrix is possible in Cantor using the array object. All indexing operations performed on matrices are mapped into the corresponding *put* and *get* messages for the array object along with the necessary synchronization. Although the put and get messages are sufficient for manipulating matrices, better use of a matrix can often be achieved by providing the matrix with the capability to react to many different kinds of messages. As an example of this approach, consider the problem of solving a set of m linear equations with n unknowns using the technique of Gaussian elimination applied to an m by n matrix. To solve for the unknowns, the matrix is first placed into reduced row-echelon form and then back substitution is used to output the computed values for the unknowns. The program shown in Figures 3.19 and 3.20 performs the row reduction phase with pivoting included to minimize round-off error. The algorithm consists of the following steps with j ranging from 1 to m:

1. Find pivot row for column j.
2. Move pivot row to the top of the matrix.
3. Adjust pivot row so that value in column j is 1.
4. Add multiples of the top row to the rows beneath it so column j is zero.
5. Cover top row, increment j and repeat.

A cursory examination of this program indicates that the total number of arithmetic operations performed is $O(m^2n)$. Step 4 can be performed concurrently, reducing the time to completion by a factor of $O(m)$. Thus the total time to completion would be $O(mn)$ instead of $O(m^2n)$ as for a sequential computer.

The program of Figures 3.19 and 3.20 organizes the Gaussian elimination algorithm as operations that can be performed on the individual elements of the matrix (Figure 3.19) and as objects that represent the first four steps listed above (Figure 3.20). The main object is used to both define the matrix and send the gauss object a message containing a reference to the matrix object element in the upper left corner of the matrix. The matrix could have been constructed dynamically as was done for the array object of Chapter 2, however to keep the example simple, it was defined statically inside of main.

The gauss object accepts a reference to the upper left corner element and checks whether there are any equations or unknowns left. If no equations or unknowns are remaining, the original upper left corner of the matrix is sent a "print" message to output the matrix in reduced row-echelon form. Otherwise, the first four steps of the above algorithm are accomplished as the composition of the three objects: compare, exchange, and subtract. The compare object finds the pivot row for the current column. The exchange object swaps

```
element (next_row, next_col, content) ::
*[ case (cmd) of
   "print" : (output, back)
             send (content) to output
             if back = "nil" then back := next_row fi
             if next_col = "nil"
                then send () to output
                     if back <> "nil"
                        then send ("print",output,"nil") to back
                     fi
                else send ("print",output,back) to next_col
             fi
"find piv" : (caller)
             send(content, self) to caller
             if next_row <> "nil"
                then send ("find piv", caller) to next_row
             fi
    "swap" : (max_val, swap_row, reply_to)
             content := content / max_val
             send ("set 1st", content, self, max_val, reply_to) to swap_row
 "set 1st" : (new_val, swap_row, max_val, reply_to)
             send ("set 2nd", content, next_col, max_val, reply_to) to swap_row
             content := new_val
 "set 2nd" : (new_val, swap_row, max_val, reply_to)
             content := new_val
             send () to reply_to
             if next_col <> "nil"
                then send ("swap", max_val, swap_row, reply_to) to next_col
             fi
  "set mf" : (next_top_col, done)
             if next_top_col = self
                then if next_row <> "nil"
                        then send ("set mf", next_col, done) to next_row
                     fi
                     send () to done
                else send ("mpy", content, next_col, done) to next_top_col
                     if next_row <> "nil"
                        then  send ("set mf", next_top_col, done) to next_row
                     fi
                     content := 0.0
             fi
     "mpy" : (mf, add_col, done)
             send ("sub", mf, content, next_col, done) to add_col
     "sub" : (mf, val, next_top_col, done)
             content := content - mf * val
             if next_top_col = "nil"
                then send () to done
                 else send ("mpy", mf, next_col, done) to next_top_col
             fi
"next col" : (next_gauss) send ("next row", next_gauss) to next_col
"next row" : (next_gauss) send (next_row) to next_gauss
]
```

Figure 3.19: Matrix Element for Gaussian Elimination Program

```
subtract (matrix, count, gauss, output) ::
*[ ()
   count := count - 1
   if count = 0 then send ("next col",gauss) to matrix fi
 ]

exchange (matrix, count, sub) ::
*[ ()
   count := count - 1
   if count = 0 then send ("set mf", matrix, sub) to matrix fi
 ]

compare (max_val, max_row, count, matrix, exch) ::
*[ (test_val, test_row)
   count := count - 1
   if (max_row = "nil") or ((abs max_val) < (abs test_val))
      then max_val := test_val
           max_row := test_row
   fi
   if count = 0 then send ("swap", max_val, matrix, exch) to max_row fi
 ]

 gauss (orig_matrix, m, n, output) ::
*[ (matrix)
   if (m = 0) or (n = 1)
      then send ("print",output,"nil") to orig_matrix
      else let sub  = subtract(matrix, m, self, output)
           let exch = exchange(matrix, n, sub)
           let cmp  = compare (0.0, "nil", m, matrix, exch)
           send ("find piv",cmp) to matrix
           m := m - 1
           n := n - 1
   fi
 ]

[ (console)
  let m = 4  % m rows by
  let n = 5  % n columns
  let m11 = element(m21,  m12,  3.0)  let m21 = element(m31, m22,   1.0)
  let m12 = element(m22,  m13,  1.0)  let m22 = element(m32, m23,   1.0)
  let m13 = element(m23,  m14,  7.0)  let m23 = element(m33, m24.   4.0)
  let m14 = element(m24,  m15,  9.0)  let m24 = element(m34, m25,   4.0)
  let m15 = element(m25, "nil", 4.0)  let m25 = element(m35, "nil", 7.0)

  let m31 = element(m41, m32,  -1.0)  let m41 = element("nil", m42,  -2.0)
  let m32 = element(m42, m33,   0.0)  let m42 = element("nil", m43,  -1.0)
  let m33 = element(m43, m34,  -2.0)  let m43 = element("nil", m44,  -4.0)
  let m34 = element(m44, m35,  -3.0)  let m44 = element("nil", m45,  -6.0)
  let m35 = element(m45, "nil", 0.0)  let m45 = element("nil", "nil", 6.0)

  send (m11) to gauss(m11, m, n, console)
]
```

Figure 3.20: Gaussian Elimination Program

the pivot row with the current top row and also does step 3 as part of the swap operation. The subtract object performs step 4.

After the gauss object builds the three additional objects, it sends a "find piv" message to the upper left corner of the matrix with a reference value of the compare object as the contents of the message. The gauss object then decrements the number of rows and columns in the matrix. The next message the gauss object accepts will be the upper left hand corner matrix object for the matrix that is one row and column smaller than the current matrix.

When the matrix accepts the "find piv" message, it sends the contents of the matrix element to the compare object and forwards the message along the first column of the matrix. The compare object keeps track of the matrix object for the upper left corner of the matrix and of the matrix object for the row whose element in the first column is the largest. When the compare object has received m messages, all of the rows of the matrix have sent a value for column 1. The compare object then sends a "swap" message to the row with the maximum element value and an argument of the first row of the matrix. The "swap" message causes a sequence of "set 1st" and "set 2nd" messages to thread (cross-stitch) their way along the two rows, exchanging the elements on a column by column basis and adjusting the contents of the pivot row as described in step 3. For every stitch there is a message sent to the exchange object. After the exchange object has received n messages from the stitch operation, the upper left corner of the matrix is sent a "set mf" message to set the multiply factor for the first column.

When the element object of the matrix receives the "set mf" message, it will send a "mpy" message to self to start the multiply and subtract step for the current row with the pivot row. A check is first made though to make sure the multiply and subtract step is not applied to the pivot row. The element object will also forward the "message" to the next row of the matrix, thereby allowing the multiply and add steps to proceed concurrently for all the rows. The "mpy" and "sub" messages work in a fashion similar to the cross-stitching of the messages used to accomplish the row exchange step.

For each completed multiply and subtract step performed on a row, a message is sent to the subtract object. After the subtract object has received m completion messages, the next iteration of the algorithm is started by first computing a new upper left corner element object for the matrix and then sending the reference value for the new corner element to the gauss object. The corner element is computed by sending the current corner element a "next col" message. The "next col" messages will cause the element object to send a "next row" message to the next column of the matrix, which in turn will send the reference value for the next row of the matrix to the gauss object.

The plots for the object count and concurrency index are shown in Figures 3.21 and 3.22. To make the plots more interesting and readable, the number of equations was increased from 4 to 64 and the printing phase of the program was eliminated. The object count is not interesting, the number of objects grows to approximately 4096, i.e. a 64 by 64 array of element objects, and remains at this level for the duration of the program.

The concurrency index is quite interesting. For the find pivot, move pivot row, and adjust pivot row steps, there is no concurrency. For the add multiples of top row to rows beneath top row, the concurrency index is equal to the number of rows. By the iterative nature of the program, as the matrix becomes one row and one column smaller, the Concurrency Index decreases by one. Likewise the interval between the concurrent "phases" is also decreasing by 1. Therefore as the computation progresses, the concurrency steadily and monotonically decreases and the interval between concurrent actions also decreases steadily and monotonically.

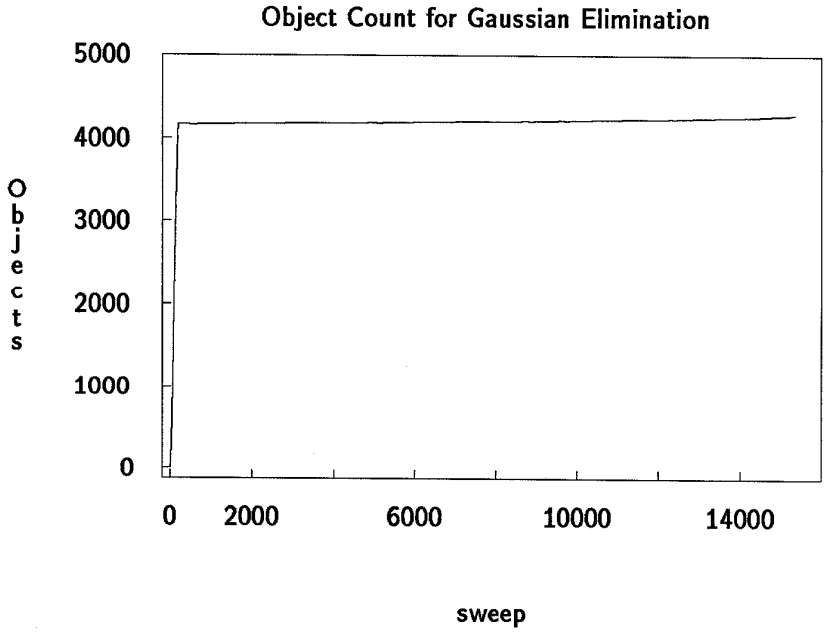

Figure 3.21: Object Count for Gaussian Elimination ($N = 64$)

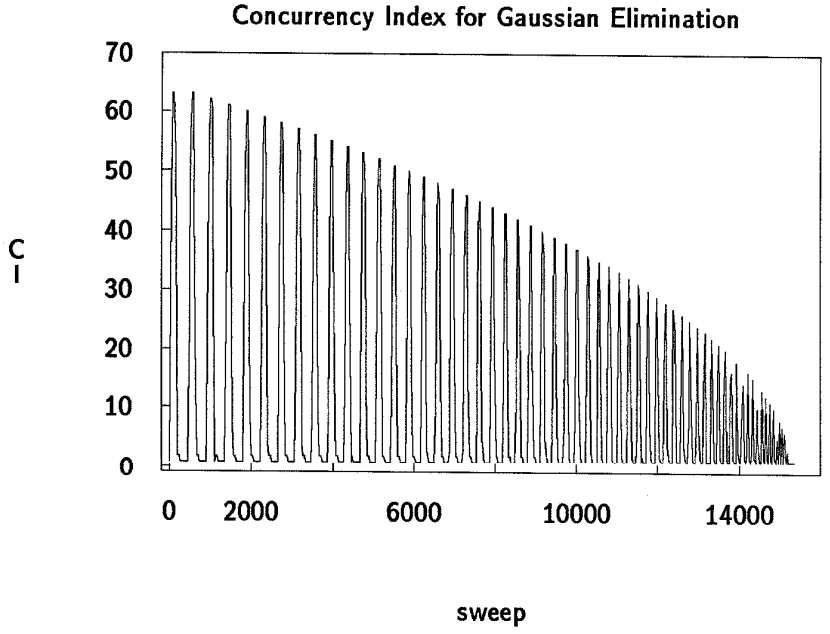

Figure 3.22: Concurrency Index for Gaussian Elimination ($N = 64$)

Chapter 4

The Cantor Programming Environment

This section documents the tools currently available for developing Cantor programs for both concurrent and sequential computers. These tools consist of a compiler, code generator, various interpreters, and a library of example programs including all the Cantor programs mentioned in this report.

The following explanations of the "nuts and bolts" for using the Cantor tools presumes the use of the Berkeley Unix operating system as well as access to the cosmic environment [9] for executing Cantor programs on a concurrent computer.

4.1 Compiling and Code Generation

The Cantor compiler is called `cfe` and is invoked by typing:

```
cfe prog
```

where `prog.can` is the name of the Cantor source text. The file extension ".can" is added to the file name input to `cfe`. As the compiler compiles each object definition, the name of the object is printed on standard output. The last object name to be printed is always "_main". This name is reserved by the compiler for the main object and should never be used by the program writer. Error messages from the compiler are sent to standard output and are fairly informative. At least they are better than `cc`.

After a program has successfully compiled, the output file `prog.Imf` is written on the same directory where `prog.can` was found. The `.Imf` extension is an abbreviation for "intermediate format". The `.Imf` file is used for both future optimization programs and for immediate code generation. To generate code suitable for execution by the current bevy of Cantor interpreters, the `.Imf` file has to be processed by the code generation program called `cgen`. The `cgen` program is invoked by typing:

```
cgen prog
```

`cgen` will automatically append the `.Imf` file extension. The output file written by `cgen` is `prog.cg3`. The `.cg3` file extension signifies that the third generation code generator format is the current format in use.

4.2 Executing Programs on a Sequential Computer

The sequential interpreter is called `csr` and can be installed on any computer that supports a C compiler. The interpreter requires a `.cg3` file from the code generation program

and is invoked by typing:

```
csr prog
```

As before, the correct file extension is added by the program. All output from `csr` is sent to standard output.

There are two "instrumented" versions of `csr` called `dsr` and `tsr`. The interpreter `dsr` is a non-interactive debugger that steps through the execution of each object. The display of an executing object consists of the definition name for the object, its reference value, the list of persistent variables for the object, and the list of message values for the object. Each new object created and new message sent as a result of executing the object is also displayed. The interpreter `tsr` maintains frequency counts for the various intermediate format instructions used during the execution of a program. The table of instruction counts is written to standard output when the program terminates. The table is of interest to the program writer since quantities such as the number of arithmetic instructions (ALU) can be compared with the number of message passing operations (SEND).

4.3 Executing Programs on a Concurrent Computer

Cantor programs can be executed concurrently on either the Caltech Cosmic Cubes or the Intel iPSC [7]. Essentially the interpreter is separated into two parts called `csr_host` and `csr_node`. The program `csr_host` is the only program the programmer needs to know about. The `csr_host` program will handle all arbitration for allocating a cube, distributing the Cantor program, and handling all the output from the cube. The cosmic environment must be present for `csr_host` to function. More information about the cosmic environment can be found in a "The C Programmer's Guide to the Cosmic Cube" [9].

The `csr_host` program can be invoked the same way as `csr`, but two optional parameters are also available. The command syntax is:

```
csr_host filename [cube dimension] [slack]
```

The optional parameter `[cube dimension]` allows the person running the Cantor program to dial in a desired size for the cube. For highly concurrent Cantor programs, the bigger the cube size, the faster the program will run, though the likelyhood of successfully allocating a cube decreases with increasing size in a space sharing environment. The default cube size is 3. The parameter `[slack]` determines the number of messages each node can have outstanding in the message system. The minimum slack is 1, else a program would not run. Performance increases dramatically when the slack is increased from 1 to 2, with diminishing returns thereafter. For many programs slack is not an issue. For some computations such as the concurrent Eight Queen's program of Chapter 3, too much slack can overload the message system of the concurrent computer. The default slack is 5. An example of a complete invocation for `csr_host` is:

```
csr_host prog 6 10
```

This invocation would be interpreted as: execute the program prog on a 6-cube with slack set to 10.

4.4 Program Termination and Timing

Program termination is detected when all internal message queues become empty. For all of the programs used in the report, the quiescent state is when there are no messages remaining in the program. The empty message queue condition is easy to detect for the sequential interpreters since there is nominally only one message queue. Once the message queue becomes empty, the interpreter program will exit normally. The concurrent interpreters employ an algorithm by Dijkstra and Scholten [5] to detect when all of the message queues inside the cube have become empty and when all of the node interpreters have become idle. After the cube has signalled the `csr_host` program, via node 0, that the computation has terminated, the `csr_host` program will terminate all node processes, release the cube, and then exit normally.

Both the concurrent and sequential interpreters keep track of the time interval between the acceptance of the first Cantor message and when all of the message queues have become empty. For the concurrent interpreters, the time interval is between when node 0 receives its very first message from `csr_host` and when node 0 signals `csr_host` that the computation has finished. Thus the time measured is the elapsed time for executing the Cantor program without considering the time spent in setting up the Cantor program, e.g. loading the programs and initializing the heaps. The elapsed time is written to standard output upon normal termination. The time units are seconds and the resolution varies with machines. For the Cosmic Cubes the resolution is 2ms while for the iPSC and VAXes it is 16ms. Resolution for SUN's varies with CPU.

4.5 Error Reporting

Runtime errors can occur either from exhausting physical resources such as heap space or object name tables, or from programming errors that the compiler was unable to detect, most notably type mismatch. For example, an object definition such as:

```
sum (time) :: [ (x,y) send (x+y) to time ]
```

cannot always be checked for proper type usage when the program is compiled. For example, if x or y was a reference value, a runtime error would result.

Whenever a runtime error occurs, the name of the object definition that caused the error is sent to standard output along with a brief message about the cause of the error. The following summary is a partial list of the possible runtime error messages followed by a short explanation of what the cause of the error might be:

`Predicate of 'if' must be type Boolean, not type ...`

The expression part of the if statement was evaluated and did not yield a logical (Boolean) value. Check the if statements inside the name of the object definition specified in the error message.

`Type incompatibility in ALU instruction.`

An attempt was made to evaluate an arithmetic or logical expression but the operands of the expression were not of compatible types. The following is a table of the allowable application of operators to data types:

operators	*data types*
=, ⟨⟩	(any,any)
+, -, *, /	(int,int) (real,real) (int,real)
⟨, ⟩, ⟨=, ⟩=,	(int,int) (real,real)
and, or, xor	(bool,bool) (int,int)
not	(bool) (int)
mod	(int,int)
abs	(int) (real)

`Heap space exhausted.`

The storage local to a node was exhausted. A node will run out of storage if there are either too many messages or too many objects. Many times such problems can be solved by changing the message flow strategy, for example, as in the prime's sieve programs of Chapter 3.

`An attempt was made to execute a deallocated object.`

A message was sent to an object that self-destructed. Fortunately the object identification number was not recycled before the bad message arrived; otherwise this error would affect the new object. Often this problem arises by forgetting to put an asterisk in front of a description, i.e. *[...] versus [...].

`Object table exhausted.`

Either too many objects were created from a single node or the program requires more objects than the system can assign names to. The first case often results from bad load balancing, not directly the programmer's fault, but often can be fixed by redistributing the task of object creation over more objects in a more uniform fashion.

4.6 External Objects

At present only two external objects exist for Cantor: console and chessboard. Any message sent to console will be displayed prefixed by the string "`console>`" on the same device that is running the interpreter.

The chessboard object exists only for the concurrent interpreter `csr_host` when executed on a SUN workstation in the Caltech C.S. Department. The program `chess_host` must be started immediately after the `csr_host` program begins. The `chess_host` program attempts to enter the cosmic environment as a second host program. If the program is started before `csr_host`, it will fail to enter the cosmic environment and terminate under an error condition. For best results, `chess_host` should be run in a separate window, preferably a `graphictool` instead of a `shelltool`. The protocol for displaying a chessboard configuration is to send eight coordinate pairs followed by an empty message to chessboard. The chessboard display will hold the configuration until a new configuration is accepted. Every configuration will be displayed for at least 5 seconds.

Chapter 5

Synopsis

This chapter provides a summary of the Cantor syntax. This chapter can be used as a quick reference for comprehending Cantor programs.

5.1 Names

All names used in Cantor programs begin with an alphabetic character followed by zero or more alphanumeric characters from the following set:

```
abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ_0123456789
```

Names are delimited by either blank space or an "end of line" character. All alphabetic characters are converted to lower case. Names are used to identify variables and object definitions but some names are "reserved" by Cantor. The following is a list of names that are special to Cantor and should not be used by the programmer.

send	to	if	then	else	fi
let	case	of	exit	repeat	become
or	xor	and	not	mod	abs
self	true	false			

The comment character for Cantor is percent sign (%). Whenever a percent sign occurs, all characters between the percent sign and the "end of line" character are ignored.

5.2 Data Types

Integer 32-bit integer number between −2147483648...2147483647 inclusively.

Real 32-bit (single precision) floating point number. Accuracy of mantissa and integral are machine dependent. The format for real numbers is an optional minus sign, followed by a string of one or more digits, followed by a decimal point, and concluded by a second string of zero or more digits.

Symbol String of alphanumeric characters enclosed between double quotes. A symbol may include any printable character including blank spaces. Symbols are compiled as constants. The only operation defined on symbols is equality testing. Examples: "Hello", "Hello from Cantor", "Hee-Yuck", "Hee Yuck".

Logical Binary or Boolean value, textually denoted as true or false.

Reference Uniquely identifies a Cantor object. The only operation defined on references is equality testing. References are generated either by the keyword self or by creating new objects using the following syntax: ⟨*object name*⟩ ⟨*list*⟩. A list is delimited by matching parentheses and contains a sequence of zero or more expressions that are separated by commas.

5.3 Expressions

Expressions are evaluated from left to right based upon the follow precedence structure for the arithmetic and logical operators.

not	abs	-					(highest — unary)
*	/	mod					(highest — binary)
+	-						
=	⟨⟩	⟨	⟩	⟨=	⟩=		
and							
or	xor						(lowest)

All operators may not be applied to all data types. The following table expresses the compatibilities between data types and operators.

operators	*data types*
−, ⟨⟩	(any,any)
+, -, *, /	(int,int) (real,real) (int,real)
⟨, ⟩, ⟨=, ⟩=,	(int,int) (real,real)
and, or, xor	(bool,bool) (int,int)
not	(bool) (int)
mod	(int,int)
abs	(int) (real)

5.4 Statements (Commands)

send (*list*) to *destination*

Send a message to the object whose reference is the value of the expression *destination*. *list* is a sequence of expressions that are separated by commas. Each expression of *list* is evaluated and made into a component of the message.

if *predicate* then *truepart* else *falsepart*

The expression *predicate* is evaluated to yield a logical value, else a runtime error occurs. If the value of *predicate* is true then the statements contained in *truepart* are executed. If the value of *predicate* is false, then the statements contained in *falsepart* are executed. The keyword else and the statements in *falsepart* are optional for the if statement.

let *letname* = *expression*

letname is created as a temporary variable inside the current description. *expression* is evaluated and the resulting value is assigned to *letname*. If *expression* evaluates to a new object reference, the reference value is assigned to *letname* before *expression* is completely evaluated. Thus *letname* may be used inside of *expression*. Furthermore, *letname* may be used anywhere within the contiguous block of let statements where it is defined.

varname := *expression*

varname is the name of either an acquaintance or message variable. *expression* is evaluated and the resulting value is assigned to *varname*.

exit

Processing of the current message is terminated and the current description for processing messages is exited. If the exit command occurs in the outermost description, the object will self-destruct.

repeat

Processing of the current message is terminated and the current description will be reused for processing the next message.

become *other*

Processing of the current message is terminated. The current object is destroyed and all future messages are automatically forwarded to *other*. The expression *other* must produce a reference value.

[(*messagelist*) *sequence*]

Processing of the current message is terminated. The object will use *messagelist* and *sequence* to process the next message. *messagelist* is a list of variable names, separated by commas, that is used to identify the contents of the next message received. *sequence* is a sequence of statements to be executed when the message arrives. If an asterisk (*) is placed in front of [, the description will be continually reused for each message received; otherwise the description will be used only once.

```
[ case ( dispatch ) of
  select1 : (messagelist) sequence
  select2 : (messagelist) sequence
  .       .               .
  .       .               .
  .       .               .
  selectN: (messagelist) sequence
]
```

Processing of the current message is terminated. The next message received will be processed using the case description. The first component of the next message received is assigned to the message variable *dispatch* and is compared with *select1* through *selectN*, which must be symbol, integer, or logical constants. If a match is found, the corresponding *messagelist* is used to identify the rest of the message contents and the corresponding *sequence* is used to process the message. If an asterisk (*) is placed immediately in front of [, the case description will be continually reused for each message received; otherwise the case description will be executed only once.

5.5 Object Definitions

A program is a collection of object definitions followed by a single description for the main object. An object definition has the following syntax:

⟨*object name*⟩ ⟨*acquaintance list*⟩ : : ⟨*description*⟩

⟨*acquaintance list*⟩ is a list of variables names, separated by commas and enclosed in parentheses. The acquaintance list defines the *persistent* variables for the object. The various forms of ⟨*description*⟩ are described in Section 5.4.

Bibliography

[1] H. Abelson and G.J. Sussman, *Structure and Interpretation of Computer Programs*, The MIT Press, Cambridge, Mass., 1985.

[2] G.A. Agha, *Actors: A Model of Concurrent Computation in Distributed Systems*, MIT Artificial Intelligence Laboratory, Technical Report 844, June 1985.

[3] W.C. Athas, *XCPL: An Experimental Concurrent Programming Language*, Dept. of Computer Science, California Institute of Technology, Technical Report 5196, Dec. 1986.

[4] E.W. Dijkstra, *A Discipline of Programming*, Prentice-Hall Inc., Englewood Cliffs, N.J., 1976.

[5] E.W. Dijkstra and C.S. Scholten, *Termination Detection for Diffusing Computations*, Philips Research Laboratories, EWD687, Eindhoven, The Netherlands, Oct. 1978.

[6] R.K. Guy, *How to Factor a Number*, Proceedings of the Fifth Manitoba Conference on Numerical Mathematics, Utilitas Mathematics Publishing Inc., Winnipeg, Oct. 1975.

[7] Intel Scientific Computers, *iPSC User's Guide*, Order No. 175455-001, 15201 N.W. Greenbrier Parkway, Beaverton, Oregon, Aug. 1985.

[8] K. Jensen and N. Wirth, *Pascal User Manual and Report*, Springer-Verlag, New York, 1974.

[9] W-K. Su, R. Faucette and C.L. Seitz, *The C Programmer's Guide to the Cosmic Cube*, Dept. of Computer Science, California Institute of Technology, Technical Report 5203, Sept. 1985.